Types for Location and Data Security in Cloud Environments

Gazeau, Ivan; Chothia, Tom; Duggan, Dominic

DOI:
10.1109/CSF.2017.25

Document Version
Peer reviewed version

Citation for published version (Harvard):
Gazeau, I, Chothia, T & Duggan, D 2017, Types for Location and Data Security in Cloud Environments. in *2017 IEEE 30th Computer Security Foundations Symposium (CSF).* IEEE Computer Society Press, 30th IEEE Computer Security Foundations Symposium, Santa Barbara, United States, 21/08/17.
https://doi.org/10.1109/CSF.2017.25

Types for Location and Data Security in Cloud Environments

Ivan Gazeau
LORIA, INRIA Nancy - Grand-Est
France

Tom Chothia
School of Computer Science,
Univ. of Birmingham, UK

Dominic Duggan
Stevens Inst of Technology,
Hoboken, NJ

***Abstract*—Cloud service providers are often trusted to be genuine, the damage caused by being discovered to be attacking their own customers outweighs any benefits such attacks could reap. On the other hand, it is expected that some cloud service users may be actively malicious. In such an open system, each location may run code which has been developed independently of other locations (and which may be secret). In this paper, we present a typed language which ensures that the access restrictions put on data on a particular device will be observed by all other devices running typed code. Untyped, compromised devices can still interact with typed devices without being able to violate the policies, except in the case when a policy directly places trust in untyped locations. Importantly, our type system does not need a middleware layer or all users to register with a preexisting PKI, and it allows for devices to dynamically create new identities. The confidentiality property guaranteed by the language is defined for any kind of intruder: we consider labeled bisimilarity i.e. an attacker cannot distinguish two scenarios that differ by the change of a protected value. This shows our main result that, for a device that runs well typed code and only places trust in other well typed devices, programming errors cannot cause a data leakage.**

I. Introduction

Organisations commonly trust their cloud providers not to be actively malicious but may still need to verify that the cloud service does not make mistakes and does store their data at particular locations. For example, data protection laws prevent the storage of certain forms of data outside the European Union. In order to ensure compliance, a company policy may require that data from a user's device not be synchronized with a cloud storage provider, unless that provider can certify that the data will not be stored in data centers outside the EU. Such checks against inappropriate storage of data can be costly and time consuming, sometimes leading to organisations not allowing their employees to use cloud services, even though a particular cloud service may be known to be compliant with data handling policies.

This paper presents a language-based approach to dealing with these kinds of scenarios, of ensuring that data can be shared with cloud services while ensuring compliance with policies on data storage. The approach is based on a type system that explicitly models trust between cloud services and mobile devices, based on a notion of principals represented at runtime by cryptographic keys. In this language, principals are dynamically associated to (non-disjoint) sets of devices, and the rights of devices to access data is based on sets of principals (delineating the principals that are allowed to access protected data). For instance, if two devices A and B can (individually) act for a principal P, while devices B and C can act for principal Q, then the right $\{P, Q\}$ implicitly allows the devices A, B and C to access data guarded by this right. We argue that this two-layer representation of access rights (data guarded by principal rights, and devices acting for principals) is convenient to allow principals to share a device, and to allow one principal to use several devices (laptop, mobile phone etc.)

Based on this type system, we present a language that includes most of the primitives necessary for secure imperative programming (multi-threading, references, secure channel establishment, and cryptographic ciphers). A key feature of this language is the ability for new principals to dynamically join the network (in the sense of making network connections to cloud services and other devices) without having to register to any public key infrastructure (PKI) or to use a particular middleware layer[1].

Our threat model assumes some "honest" collection of principals (e.g. the employees of an enterprise) and some collection of devices acting for those principals (e.g. devices provided to those employees). A device may act for several principals, in the sense that it may issue access requests on behalf of any of the principals that it acts for), while a principal may be associated with several devices. We describe the devices acting for these principals as *honest devices*, in the sense that they are certified according to the type system presented in this paper to be in conformance with data sharing policies. We refer to the corresponding principals for these devices as "honest" rather than "trusted" because trust management is an orthogonal issue for the scenarios that we consider. Our threat model also allows for "dishonest" or untyped devices, acting for outside principals, who should not be able to access the data. These are the attacker devices. Our security guarantee is that, as long as no honest device provides access to a dishonest principal, the dishonest devices will not be able to obtain any information from any honest devices, unless an honest device has explicitly given the attacker access.

This capability is critical for cloud services. While it is reasonable to assume that there exists a PKI to certify the identities of cloud providers, and that cloud providers are trusted by their users, client devices and their corresponding principals are unlikely to have certificates. While an infrastructure for mutual authentication, based on client and server X509 certificates, might be provided as part of an enterprise data sharing network, there are difficult issues

[1] Although middleware for e-commerce (such as CORBA) was popular in the 1990s, that approach was discredited by experience, while SOAP-based approaches have been largely superseded by REST-based Web services, that deliberately eschew the notion of a middleware at least for Internet communication..

with extending this trust model to third party cloud service providers. Furthermore, such a PKI does not provide the level of confidence in conformance with data-sharing protocols, that our approach provides for honest (well-typed) devices. Instead of requiring such a global PKI, the approach described in this article represents principals by cryptographic public keys, and these are stored on devices like any other values. Our type system therefore uses a nominal form of dependent types, to reflect these runtime representatives of principal identity to the type system, where access rights on data are tracked through security labels attached to types.

Another contribution of the paper is a security guarantee suitable for our threat model. As stated above, in an open network, we must allow for "dishonest" devices that are unchecked (untyped), and potentially malicious. These devices are able to use any third-party cloud services, including services used by honest devices. A secure data-sharing system should remain robust to such an intruder. Our security guarantee is that if data is protected with rights that only includes honest principals (i.e., that do not include any principal which is associated to an attacker device), then an attacker cannot learn any information about that data. In this work, we are focused on confidentiality of the data, and do not consider integrity (that data has not been tampered with by attackers). There is a notion of integrity in the sense of trust management underlying our approach: Honest principals identify themselves by their public keys, and these keys are to state access restrictions on data, to specify when a device is allowed to "act for" a principal, and to validate that communication channels are with honest parties trusted to be type-checked and therefore conformant with data-sharing policies. This is reflected in several aspects of the communication API, including the establishment of communication channels, generating new principals and transmitting those principals between devices.

A practical difficulty with expressing security guarantees in this setting is that, as principals can be created dynamically, it is not possible to statically check if a variable has a higher or lower security level than some other variable. Consider the following example:

Example 1: Assume three devices: an honest cloud service with a certified principal p, and two (mobile) devices, an honest device A and a malicious device B. The server code consists of receiving two principal values p_1 and p_2 from the network before it creates two memory locations x_1 and x_2 where x_1 has rights $\{p, p_1\}$ while x_2 has right $\{p, p_2\}$. A secure location is defined one which cannot be accessed by the attacker. If we assume that devices A and B send their principal values (public keys) to the server, then depending on which key is received first, either x_1 or x_2 will be considered secure (only accessible by the honest principal and the cloud provider), while the other is explicitly accessible by the attacker. As usual with information flow control type systems, we propagate these access restrictions through the handling of data by the cloud provider and the devices, ensuring that data protected by the right $\{p, p_i\}$, where p_i is the representative for the honest principal.

Therefore, our security property is a posteriori: once the system has created a memory cell corresponding to some variable, if the rights associated to this variable at creation time did not include a principal controlled by the attacker, then there will never be any leak about the contents of this memory cell to the attacker. We argue that such a property is suitable for cloud services to increase users' confidence that their data will not be leaked by the service due to a programming error. Indeed, our system allows us to certify that once some principal creates data, only explicitly authorised principals can obtain information about that data, by statically checking the code that processes that data. Verifying the identity of the principals allowed to access the data, and deciding where to place trust among the principals in a distributed system, is an important consideration. However it properly remains the responsibility of the application written in our language, and a concern which is independent of this type system. Our approach serves to guarantee proper handling of data among honest principals, once an appropriate trust management system has established who is honest.

Our typed language is intended to be a low level language, without high-level notions such as objects and closures. It includes consider references, multi-threading and a realistic application programming interface (API) for distributed communication. Communications can either be through a secure channel mechanism, implementable using a secure transport layer such as TLS for example, or through public connections, in which case any device can connect. The language includes primitives for asymmetric key encryption, since we represent by public keys. "Possessing" a secret key, in the sense that the private key of a public-private key is stored in its memory, allows a device to "act for" the principal that key represents. Our approach is similar in philosophy to the Simple Public Key Infrastructure (SPKI), where principals and public keys are considered as synonymous, rather than linking principals to a separate notion of public key representatives. However, we do not include notions such as delegation that are the central consideration of SPKI, since we explicitly avoid the consideration of trust management, leaving that to applications written using the API that we provide. This also differentiates our approach from frameworks such as JIF and Fabric, that include delegation of authority to principals based on an assumed trust management infrastructure.

Nevertheless, there is a notion at least tangentially relegated to delegation of trust in our framework: In order to allow a device to act for more than one principal, our semantics allows a principal to be created on one device and communicated to another device, where it is registered on the receiver device as one of the principals upon whose behalf that device can access data. For example, a client of a cloud service provider may generate a proxy principal representing that client on the cloud service, and then upload that principal to the cloud service in order to access data that the client is storing on the cloud service. This ability to share principals across devices is controlled by restrictions established when proxy principals are generated: Such a proxy (client) principal can only be registered on a device that acts for (cloud service) principals that are identified at the point of generation of the proxy.

The security analysis of the type system uses standard techniques from the applied-pi calculus [1]. This allows us to prove our correctness property as a non-interference property based on process equivalence, i.e., two systems differing by one value are indistinguishable by any party that is not allowed to access this value. The standard pi-calculus includes message-passing

with structured values, but does not include an explicit notion of memory (although it can obviously be modeled using processes as references). Since our language combines message-passing communication and localized stateful memory, we use the stateful applied pi-calculus [2] as the starting point for our security analysis. This calculus does not explicitly model location (i.e., the distinction between two processes on the same device and two processes on two distinct devices). Since this distinction is critical for our security analysis, we add this notion in our calculus. Nevertheless the proof techniques that we employ are heavily based on those developed for the stateful applied pi-calculus.

The security analysis that we perform expresses that data are secure if keys received from other devices are not associated to an attacker. To formalise this conditional statement, we need more techniques than in a standard protocol where data are either secret or public, but their status does not depend on the execution. In our verification in Section V-C, we introduce an extended syntax that marks which keys, variables and channels are secure in the current trace. We then prove that when a new memory location is created with a secure key according to this marking, then the attacker cannot distinguish between two scenarios: one where the system reduces normally, and another one where the memory location is sometimes altered to another value. This is the basis for our noninterference property for the security guarantee provided by this approach. The full details of the proofs of correctness for information flow control are provided in the complete version of the paper [11].

In the next section we discuss related work. In Section III we present our language, type system and semantics. In Section IV we present an extended example and in Section V present our result and outline the proof, then we conclude in Section VI.

II. Related Work

Implicit flow: Implicit information flow properties involve the ability for an attacker to distinguish between two executions. Previous work that has provided type systems to control implicit information flow [20], [3] considered high and low data, and this could be extended to a bigger lattice but not to the creation of new principals, as the security of a variable is defined statically. Zheng and Myers presented an information flow type system that includes dynamic checks on rights [22] which can be used, for instance, when opening a file. The Jif Project [17] adds security types to a subset of Java, leading to a powerful and expressive language. Unlike our work, this other work does not address how to enforce principal identities and type information to be correctly communicated to other locations.

Security properties on distributed system: Work on type security for distributed systems can be distinguished according to the kind of security they aim to provide. Muller and Chong present a type system that includes a concept of place [16] and their type system ensures that covert channels between "places" cannot leak information. Vaughan et al. look at types that can be used to provide evidence based audit [12], [19]. Fournet et al. look at adding annotations with a security logic to enforce policies [10]. Liu and Myers [14] look at a type system which ensures referential integrity in a distributed setting. This work uses a fix lattice of policy levels, which does not change at runtime. The Fabric language [13] provides decentralised, type-enforced security guarantees using a powerful middleware layer for PKI, and Morgenstern et al. [15] extend Agda with security types. In contrast, our work allows programs to generate new principals at runtime and provides a security property that tracks implicit information flow, without requiring the support of a purpose built middleware layer or global PKI. Due to the fact that the attackers in our model can access services in the same way as honest principals, this security property is an adaptation of the bisimulation property which is a strong property introduced in the (s)pi-calculus by [1]. Bisimilation can be checked for processes by tools like Proverif [5] but these kind of tools do not scale up to large systems.

Managements of new principals: Bengtson et al. [4] present a security type system which allows creation of dynamic principals in presence of an untyped attacker. However, this type system provides only assertion-based security properties of cryptographic protocols. These are weaker than non-interference properties as they are expressed on one process instead of comparing two processes. [7] considers a framework in which principals can be created at run time (without a global PKI) they prove type soundness rather than a non-interference result. Finally, the DSTAR program [21] achieves these two goals but is focused on network information and relies on local systems to actually analyse implicit flow, which leads to a more coarse system.

Safety despite compromised principals: Past work [6] has looked at un-typed attackers in a security type system, however this work only considers a static number of principals, fixed at run time. Fournet, Gordan and Maffeis [9] develop a security type system for a version of the applied pi-calculus extended with locations and annotation. Their type system can enforce complex policies on these annotations, and they show that these policies hold as long as they do not depend on principals that have been compromised. Unlike our work they assume that the principals are all known to each other and there is a direct mapping from each location to a single principal that controls it. Our work allows principals to be dynamically created, shared between locations and for locations to control multiple principal identities. We argue that this model is a better fit to cloud systems in which users can dynamically create many identities and use them with many services.

III. Language: Semantics and Type System

A. Syntax

The syntax of our language is given in Figures 1, 2, 3 and 4. We let x, y, z range over variable names $\mathcal{N}_v$, $p, p_1, p_2, \ldots$ range over principal names $\mathcal{N}_p$, $k_1, k_2, \ldots$ range over public key names $\mathcal{N}_k$ and $c, c_1, c_2, \ldots$ range over channel names $\mathcal{N}_c$. A system $\nu\bar{c}.D_1 \mid \ldots \mid D_n$ is a set of *devices* that run in parallel and that communicates through channels of $\bar{c}$.

The list $\bar{c}$ records which channel names correspond to establish channels (globally bound). Channel also appear in connect and accept commands in the devices, and these are added to $\bar{c}$ once the channel is opened. When there are no established channels, we omit the $\nu\{\}.$ prefix. Note that to guarantee freshness of keys and nonce used in encryption,

P, Q	::=	$\nu \bar{c}.D$	devices sharing channels $\bar{c}$
D	::=	$\langle M \blacktriangleright C \rangle \mid D$	C running with M
	|	0	no device
M	::=	$\{\}$	a memory which maps ...
	|	$\{x \mapsto v\} \cup M$	...a variable
	|	$\{p \mapsto P\} \cup M$	...a principal name
	|	$\{k' \mapsto k^+\} \cup M$	...a key name
v	::=	i	an integer
	|	k^+	the value of a public key
	|	$\mathbf{enc}_{LR,n}(v)$	a cipher of v with seed n
	|	$\mathbf{enc}_{LR,n}(\mathrm{pv})$	an encapsulated principal
	|	$\{v_1, \ldots v_n\}$	an array of values
	|	NaV	a special error value
pv	::=	$\mathbf{prin}(k^+, k^-, LR)$	a principal value
LR	::=	$\{k^+\} \cup LR$	a set of public keys
	|	$\{\ \}$	
$\bar{c}$	::=	$\{\}$	
	|	$\bar{c}.c$	c : established channel

Fig. 1. The syntax of devices, values, principals and rights where k^-/k^+ is a secret/public key pair.

we might also provide global binders $\bar{k}$ and $\bar{n}$ in addition to $\bar{c}$. However this guarantee is straightforward to provide, using "freshness" predicates, so for readability reasons we omit explicit binders for generated keys and nonces.

A device consists of a *memory* M and a *command* C. Memories associate variable names with *values*, key names with keys and principal names with *principals*.

Given a nonce k from some assumed set of key nonces, we define (k^+, k^-) as the public private key pair generated from k, where $.^+$ and $.^-$ are two constructors. A principal pv is a tuple $\mathbf{prin}(k^-, k^+, LR)$ which contains a key pair (k^-, k^+), together with a (possibly empty) set of public key values LR. When a device has pv in its memory, it is allowed to act for pv. Devices that can act as a principal pv_0, whose public key ${k_0}^+$ is one of the public keys in LR, are allowed to add pv to their memory.

Each variable x in $\mathcal{N}_v$ represents a reference to a value i.e. variables are mutable. At declaration time, a reference is associated with some rights R, which cannot be revoked or changed. Making all variables mutable is convenient for our security analysis: it allows us to define non interference as the property that a parallel process that alters the value of a high-level variable cannot be detected by an attacker. If variables were not mutable we would have to consider a much more complex security property and proof. In addition, to avoid to consider scope of variables, we assume that a command never declares twice the same name for variables, channels, keys and principals.

Types (Figure 2) for these variables consists of a pure-type S which indicates the base type for the value of the variable (e.g. integer, public key, cipher, etc.) and a *label* (or *right*) R which indicates the principals who are allowed to access to the variable. A label can be either $\bot$ i.e. the variable is public or a set which contains public key names: $k \in \mathcal{N}_k$ and $\mathbf{pub}(p)$ where $p \in \mathcal{N}_p$.

Key names are declared by a command $\mathbf{let}\ k = x\ \mathbf{in}\ C$.

S	::=	$PubKey$	a public key
	|	$PrivKeyEnc$	an encapsulated private key
	|	Int	integers
	|	$Enc\{S\}$	encrypted data of type S
	|	$Array\{S\}$	an array of type S
R	::=	RS	a set of rights
	|	$\bot$	no restriction
RS	::=	$\{\ \}$	the empty set
	|	$\{K\} \cup RS$	K added to a set of rights
K	::=	$k_1, k_2, \ldots,$	public key names
	|	$\{\mathbf{pub}(p)\}$	the public key of a principal

Fig. 2. The types syntax

C	::=	$\mathbf{if}(e_1 = e_2)\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2$
	|	$\mathbf{new}\ x{:}S_R = e;\ C$
	|	$x\ :=\ e;\ C$
	|	$x[e_1]\ :=\ e_2;\ C$
	|	$\mathbf{let}\ k = x\ \mathbf{in}\ C$
	|	$C_1 \mid C_2$
	|	$\mathbf{skip}$
	|	$!\,C$
	|	$\mathbf{connect}\ c{:}\,Chan(S_\bot)_\bot;\ C$
	|	$\mathbf{accept}\ c{:}\,Chan(S_\bot)_\bot;\ C$
	|	$\mathbf{connect}\ c{:}\,Chan(S_R)_{R'}\ \mathbf{to}\ k\ \mathbf{as}\ p;\ C$
	|	$\mathbf{accept}\ c{:}\,Chan(S_R)_{R'}\ \mathbf{from}\ k\ \mathbf{as}\ p;\ C$
	|	$\mathbf{output}\ c\langle e\rangle;\ C$
	|	$\mathbf{input}\ c(x);\ C$
	|	$\mathbf{synchronized}\{G\}; C_2$
	|	$\mathbf{newPrin}\ p_{RS}; C$
	|	$\mathbf{decrypt}_p\ e\ \mathbf{as}\ x{:}S_{RS}\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2$
	|	$\mathbf{register}_{p_1}\ e\ \mathbf{as}\ p_2\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2$

Fig. 3. The syntax of the commands.

This command copies the value of the reference x into k which represents a public key (not a reference to a public key). The type system (deref_T) ensures that x in this command is an unrestricted public key: $x : PubKey_\bot$.

Channel types are declared when they are established, we have two kinds of channel: public and secure channels. Their types have syntax $Chan(S_{R_1})_{R_2}$ where S_{R_1} is the type of values that are past over the channel and R_2 expresses which principals are allowed to know the existence of c and when communication on this channel takes place.

B. Semantics

The semantics of the system is defined as a small-step semantics for commands and as a big-step semantics for expressions. Devices run concurrently with synchronized communication between them. Inside each device, all parallel threads run concurrently and communicate through the shared memory of the device (since memory is mutable). The main reduction rules are presented in Figure 5 for commands and in Figure 6 for main expressions. When a command that declares a new variable is reduced, the name is replaced by a fresh name that represents a pointer to the location in the memory where the value has been stored. The evaluation of expressions has the form $M(e) = v$: the evaluation of e with

e	::=	$x, y, \ldots$	variable names
	|	$\mathbf{pub}(p)$	the public key of p
	|	$\mathbf{release}(p)$	pack a principal
	|	$\mathbf{enc}_{RS}(e)$	encrypt some data
	|	$e_1 \oplus e_2$	where $\oplus$ is $+, -, \times, \ldots$
	|	$\{e_1, \ldots e_n\}$	an array of expressions
	|	$x[e]$	an element of an array
	|	i	an integer $i \in \mathbb{Z}$

Fig. 4. The syntax of the expressions

memory M returns the value v. We note that this is different from some other calculi, in which variables are not references, and are replaced with a value when declared. Our correctness statement below depends on the use of references and, since we have a memory mechanism, we prefer to store the key names and principal names in memory and instead of applying a substitution, the names are evaluated when a command reduces (cf the (rights_RS) rules).

Principals are generated using the $\mathbf{newPrin}\ p_{RS}; C$ command, where RS are the keys to use to protect the principal (and therefore cannot $\perp$). The rule for this command (newPrin_S) generates a fresh key pair (k^+, k^-) and stores the principal $\text{prin}(k^+, k^-, LR)$ at a new location p' in the memory, where $M(RS) = LR$. To bootstrap the creation of these principals, they can be declared with $RS = \{\}$; such principal identities can only be used on a single device, they cannot be sent over channels. Additionally some devices may start off with the public keys of some trusted parties, i.e., the same assumption as TLS. This too lends itself well to cloud systems, in which web browsers come with a number of trusted certificates.

Communication between devices uses Java like channels: a channel is first established then it can be used to input and output values. The channel establishment is done by substituting the channel names in both devices by a unique fresh channel name added to $\bar{c}$ (we assume that initial channel names and active channel names range over distinct sub-domains of $\mathcal{N}_c$ to avoid collision). Note that channels do not name the sending and receiving device as these may be spoofed by an attacker, however, to get a more tuneable system, it would be a simple extension to add a port number which would restrict possible connections. For secure channels (open_priv_S), in a similar way to TLS with client certificates, both devices must provide the public key k^+ of who they want to connect to. They must also provide the principal (which includes a private key) to identify themselves to the other party. To set up a secure channel, the client and the server also have to ensure that they are considering the same rights for the channel. For that they have to exchange the value of their channel right $M(Chan(S_{R_1})_{R_2})$ and make sure that it corresponds to the distant right value $M'(Chan(S_{R'_1})_{R'_2})$. Indeed, even if type-checking is static inside a device, type-checking has to be dynamic between distinct devices since programs are type-checked on each device and not globally.

Example 2 (Principal set up): Assume that a cloud service C has a public key c^+, which is known to Alice and Bob, and that Alice wants to share some private data with Bob using this cloud service. Alice can do this using the code shown in Figure 7. Alice starts by generating a key pair (a^-, a^+). As

$$\frac{\text{fresh}(k^+, k^-) \quad \text{fresh}(p') \qquad M(RS) = LR}{\begin{array}{c}\nu\bar{c}.D \mid \langle M \blacktriangleright C' \mid \mathbf{newPrin}\ p_{RS}; C\rangle \\ \to \nu\bar{c}.D \mid \langle M \cup \{p' \mapsto \text{prin}(k^+, k^-, LR)\} \blacktriangleright C' \mid C\{p'/p\}\rangle\end{array}} \text{(newPrin\_S)}$$

$$\frac{M(e) = v \qquad \text{fresh}(y)}{\begin{array}{c}\nu\bar{c}.D \mid \langle M \blacktriangleright C' \mid \mathbf{new}\ x{:}S_R = e;\ C\rangle \\ \to \nu\bar{c}.D \mid \langle M \cup \{y \mapsto v\} \blacktriangleright C' \mid C\{y/x\}\rangle\end{array}} \text{(new\_S)}$$

$$\frac{M(e) = v_2}{\begin{array}{c}\nu\bar{c}.D \mid \langle M \cup \{x \mapsto v_1\} \blacktriangleright C' \mid x\ :=\ e;\ C\rangle \\ \to \nu\bar{c}.D \mid \langle M \cup \{x \mapsto v_2\} \blacktriangleright C' \mid C\rangle\end{array}} \text{(assign\_S)}$$

$$\frac{\text{fresh}(k') \qquad M(e) = k^+}{\begin{array}{c}\nu\bar{c}.D \mid \langle M \blacktriangleright C' \mid \mathbf{let}\ k = e\ \mathbf{in}\ \ C\rangle \\ \to \nu\bar{c}.D \mid \langle M \cup \{k' \mapsto k^+\} \blacktriangleright C' \mid C\{k'/k\}\rangle\end{array}} \text{(deref\_S)}$$

$$\frac{\begin{array}{c}M(p) = \mathbf{prin}(k^+, k^-, LR_2) \qquad M(e) = \mathbf{enc}_{LR,n}(v) \\ k^+ \in LR \qquad M(RS) \subseteq LR \qquad \text{fresh}(y)\end{array}}{\begin{array}{c}\nu\bar{c}.D \mid \langle M \blacktriangleright C' \mid \mathbf{decrypt}_p\, e\ \mathbf{as}\ x{:}S_{RS}\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\rangle \\ \to \nu\bar{c}.D \mid \langle M \cup \{y \mapsto v\} \blacktriangleright C' \mid C_1\{y/x\}\rangle\end{array}} \text{(dec\_true\_S)}$$

$$\frac{\text{fresh}(c)}{\begin{array}{c}\nu\bar{c}.D \mid \langle M_1 \blacktriangleright C' \mid \mathbf{connect}\ c_1 : Chan(S_\perp)_\perp; C_1\rangle \\ \mid \langle M_2 \blacktriangleright C'' \mid \mathbf{accept}\ c_2 : Chan(S_\perp)_\perp; C_2\rangle \\ \to \nu\bar{c}.c.D \mid \langle M_1 \blacktriangleright C' \mid C_1\{c/c_1\}\rangle \mid \langle M_2 \blacktriangleright C'' \mid C_2\{c/c_2\}\rangle\end{array}} \text{(open\_public\_S)}$$

$$\frac{\begin{array}{c}\text{fresh}(c) \qquad M_1(p_s) = \mathbf{prin}(k_1{}^-, k_1{}^+, LR_s) \\ M_2(p_c) = \mathbf{prin}(k_2{}^-, k_2{}^+, LR_c) \\ M_1(R_1) = M_2(R_2) \qquad M_1(R'_1) = M_2(R'_2) \\ M_1(k_c) = k_2{}^+ \qquad M_2(k_s) = k_1{}^+\end{array}}{\begin{array}{c}\nu\bar{c}.D \mid \\ \langle M_1 \blacktriangleright C' \mid \mathbf{accept}\ c_1 : Chan(S_{R_1})_{R_2}\ \mathbf{from}\ k_c\ \mathbf{as}\ p_s; C_1\rangle \\ \mid\langle M_2 \blacktriangleright C'' \mid \mathbf{connect}\ c_2 : Chan(S_{R'_1})_{R'_2}\ \mathbf{to}\ k_s\ \mathbf{as}\ p_c; C_2\rangle \\ \to \nu\bar{c}.c.D \mid \langle M_1 \blacktriangleright C' \mid C_1\{c/c_1\}\rangle\mid\langle M_2 \blacktriangleright C'' \mid C_2\{c/c_2\}\rangle\end{array}} \text{(open\_priv\_S)}$$

$$\frac{M_1(e) = v \qquad \text{fresh}(y)}{\begin{array}{c}\nu\bar{c}.D \mid \langle M_1 \blacktriangleright C' \mid \mathbf{output}\ c\langle e\rangle; C_1\rangle \\ \mid \langle M_2 \blacktriangleright C'' \mid \mathbf{input}\ c(x); C_2\rangle \\ \to \nu\bar{c}.D \mid \langle M_1 \blacktriangleright C' \mid C_1\rangle \\ \mid \langle M_2 \cup \{y \mapsto v\} \blacktriangleright C'' \mid C_2\{y/x\}\rangle\end{array}} \text{(i/o\_S)}$$

$$\frac{\begin{array}{c}M(e) = \mathbf{enc}_{LR_1,n}(\mathbf{prin}(k_1^+, k_1^-, LR_1)) \\ M(p_2) = \mathbf{prin}(k_2^+, k_2^-, LR_2) \qquad k_2^+ \in LR_1 \qquad \text{fresh}(p_3)\end{array}}{\begin{array}{c}\nu\bar{c}.D \mid \langle M \blacktriangleright C' \mid \mathbf{register}_{p_2}\ e\ \mathbf{as}\ p_1\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\rangle \\ \to \nu\bar{c}.D \mid \\ \langle M \cup \{p_3 \mapsto \mathbf{prin}(k_1^+, k_1^-, LR_1)\} \blacktriangleright C' \mid C_1\{p_3/p_1\}\rangle\end{array}} \text{(register\_true\_S)}$$

$$\frac{\begin{array}{c}M(e) = \mathbf{enc}_{LR_1,n}(\mathbf{prin}(k_1^+, k_1^-, LR_1)) \\ M(p_2) = \mathbf{prin}(k_2^+, k_2^-, LR_2) \qquad k_2^+ \notin LR_1\end{array}}{\begin{array}{c}\nu\bar{c}.D \mid \langle M \blacktriangleright C' \mid \mathbf{register}_{p_2}\ e\ \mathbf{as}\ p_1\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\rangle \\ \to \nu\bar{c}.D \mid \langle M \blacktriangleright C' \mid C_2\rangle\end{array}} \text{(register\_false\_S)}$$

Fig. 5. Main command rules

$$\frac{M(p) = \mathbf{prin}(k^+, k^-, LR) \qquad LR \neq \{\} \qquad \text{fresh}(n)}{M(\mathbf{release}(p)) = \mathbf{enc}_{LR,n}(\mathbf{prin}(k^+, k^-, LR))} \quad \text{(release\_E)}$$

$$\frac{\text{fresh}(n) \qquad M(e) = v \qquad M(RS) = LR}{M(\mathbf{enc}_{RS}(e)) = \mathbf{enc}_{LR,n}(v)} \quad \text{(enc\_E)}$$

$$\frac{M(e_1) = v_1 \qquad M(e_2) = v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z}}{M(e_1 + e_2) = v_1 + v_2} \quad \text{(+\_E)}$$

$$\frac{M(e_1) = v_1 \qquad M(e_2) = v_2 \qquad v_1 \notin \mathbb{Z} \vee v_2 \notin \mathbb{Z}}{M(e_1 + e_2) = \text{NaV}} \quad \text{(error\_+\_E)}$$

$$\frac{\begin{array}{c}\forall i, 1 \leq i \leq n, \quad p_i \mapsto \text{prin}({k_i}^+, {k_i}^-, LR_i) \in M \\ \forall i, n+1 \leq i \leq n+m, \quad k'_i \mapsto {k_i}^+ \in M\end{array}}{\begin{array}{c}M(\{\mathbf{pub}(p_1), \ldots, \mathbf{pub}(p_n), k'_{n+1}, \ldots, k'_{n+m}\}) \\ = \{{k_1}^+, \ldots, {k_{n+m}}^+\}\end{array}} \quad \text{(rights\_RS)}$$

Fig. 6. Main semantic rules for expressions and rights

$$\langle k_c \mapsto c^+ \blacktriangleright \begin{array}{l} \mathbf{newPrin}\ A_{\{\}}; \\ \mathbf{accept}\ c\colon \mathit{Chan}(\mathit{PubKey}_\perp)_\perp; \\ \mathbf{input}\ c(x_b); \\ \mathbf{let}\ k_b = x_b\ \mathbf{in} \\ \mathbf{new}\ \mathit{secret}\colon \mathit{Int}_{\{k_c, \mathbf{pub}(A), k_b\}} = 42; \\ \mathbf{connect}\ \mathit{upload}\colon \mathit{Chan}(\mathit{Int}_{\{k_c, \mathbf{pub}(A), k_b\}})_\perp \\ \quad\ \mathbf{to}\ k_c\ \mathbf{as}\ \mathbf{pub}(A); \\ \mathbf{output}\ \mathit{upload}\langle \mathit{secret}\rangle; \end{array} \rangle$$

Fig. 7. Alice sharing data with Bob using a cloud service. N.B. this code does not authenticate Bob.

neither Alice nor Bob have certificates, Bob just sends his key publicly to Alice over a public channel. Alice receives it, and creates a new variable to be shared with Bob and the cloud service. She then opens a secure channel with the cloud that is typed to allow data of type $\{a^+, b^+, c^+\}$, the $\perp$ right on this channel indicates that, while the data on the channel must be kept confidential, the knowledge that some value has been sent is not. This fragment of code does not authenticate Bob, this could be done using another protocol, or offline, but we will show that if the device that sent her this key, and the cloud server, both run well typed code, then she is guaranteed that the secret will only be shared by the device sending the key, the cloud server and herself. She knows that no leak can come from, for instance, bad code design on the cloud device.

The $\mathbf{enc}_{RS}(e)$ expression, governed by the (enc_E) rule, encrypts the evaluation of e for each of the public keys ${k_i}^+$ named in RS, i.e., anyone that has a single private key corresponding to any ${k_i}^+$ can decrypt it, the set of all ${k_i}^+$ is also included in the encryption. We use randomised encryption to avoid leakage that would occur otherwise when the same value is encrypted twice, and we model this by including a fresh nonce in the encryption. The $\mathbf{decrypt}_p\ e\ \mathbf{as}\ x\colon S_{RS}\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2$ command reduces successfully (dec_true_S) when e evaluates to a ciphertext $\mathbf{enc}_{RS,n}(v)$ that can be opened by the secret key of p and that the LR, which is packed into the encryption, is a subset of the evaluation of RS.

The $\mathbf{release}(p)$ expression reduces by encrypting the principal p for each of the set of public keys representing the principals that can access it. It is the only way to produce a value which contains a secret key and therefore to send private keys through a channel. The **register** command behaves as **decrypt** except that it deals with encrypted principals instead of encrypted values.

All other semantics rules are standard except that instead of returning run-time error (division by 0, illegal offset index etc.) expression returns a special value NaV. This feature is critical to guarantee the security of our system. Indeed, we allow a device to evaluate expressions with secure variables and then to do an output on a public channel. This scenario is safe only if we can ensure that no expression will block any thread. Note that the attacker can also send values with some type through a channel of another type, consequently run time type errors can also occur.

Finally, the command $\mathbf{synchronised}\{C\}$ executes a command C with no communication or interleaving of other processes. This is useful to avoid race conditions.

C. Types

The type judgment for expressions takes the form $\Gamma_p; \Gamma_k; \Gamma \vdash e : S_R$ and the type judgment for commands takes the form $pc; \Gamma_p; \Gamma_k; \Gamma_c; \Gamma \vdash C$ where Γ is a mapping from variable names to types S_R, Γ_c from channel names to channel type $\mathit{Chan}(S_{R_1})_{R_2}$, Γ_p is a set of principal names, Γ_k is a set of public key names, and where pc is a right of form R called the *program-counter*. The program counter allows to analyse programs for indirect secure information flow [8]. Figure 8 defines the main type rules for commands and Figure 9 defines the rules for expressions and rights.

In many typing judgments, we use a condition $R_1 \subseteq R_2$ that states that R_1 is more confidential than R_2. The predicate $R_1 \subseteq R_2$ holds either when $R_2 = \perp$ or when R_1 is a syntactical subset of R_2 (no aliasing). For instance, we have $\{k_2, \text{pub}(p)\} \subseteq \{k_1, \text{pub}(p), k_2\}$ and $\{k_1, k_2\} \not\subseteq \{k_2\}$ even if k_1 and k_2 map to the same key in memory. We also use $R_1 \cap R_2$ to define the syntactic intersection of the sets R_1 and R_2 (which is R_2 if $R_1 = \perp$).

Types rules for new principals and variables: The typing rule for principal declaration (newPrin_T) only allows the program counter to be bottom. This restriction avoids the situation in which a variable with a right including this principal might be less confidential than the principal itself.

The new variable declaration (new_T) checks that the rights R_1 to access the new variable x are more restrictive than (a subset of) the rights R_2 of the expression being assigned and of the program counter pc. We also ensure that one of the principal in R_1 belongs to Γ_p. The type rule for assignment (assign_T) ensures that high security values cannot be assigned to lower security variables.

While the semantics for new principals (newPrin_S) stores the rights set LR dynamically (as LR is only used when the principal is sent to another device), the semantics for managing

$$\frac{\Gamma_p;\Gamma_k \vdash RS \qquad pc = \bot \qquad pc;\Gamma_p \cup \{p\};\Gamma_k;\Gamma_c;\Gamma \vdash C}{pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash \mathbf{newPrin}\ p_{RS};C} \text{(newPrin\_T)}$$

$$\frac{\begin{array}{c}\Gamma_p;\Gamma_k \vdash R_1 \qquad \Gamma_p;\Gamma_k;\Gamma \vdash e : S_{R_2} \\ R_1 \subseteq pc \cap R_2 \qquad \exists p \in \Gamma_p.\mathbf{pub}(p) \in R_1 \\ pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \cup \{x : S_{R_1}\} \vdash C\end{array}}{pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash \mathbf{new}\ x{:}S_{R_1} = e;C} \text{(new\_T)}$$

$$\frac{\begin{array}{c}\Gamma_p;\Gamma_k;\Gamma \vdash x : S_{R_1} \quad \Gamma_p;\Gamma_k;\Gamma \vdash e : S_{R_2} \\ R_1 \subseteq pc \cap R_2 \quad pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash C\end{array}}{pc;\Gamma_p;\Gamma_k;\Gamma \vdash x\ :=\ e;\ C} \text{(assign\_T)}$$

$$\frac{\begin{array}{c}pc = \bot \\ \Gamma_p;\Gamma_k;\Gamma \vdash x : PubKey_\bot \qquad pc;\Gamma_p;\Gamma_k \cup \{k\};\Gamma_c;\Gamma \vdash C\end{array}}{pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash \mathbf{let}\ k = x\ \mathbf{in}\ C} \text{(deref\_T)}$$

$$\frac{\begin{array}{c}\Gamma_p;\Gamma_k;\Gamma \vdash e_1 : S_{R_1} \\ \Gamma_p;\Gamma_k;\Gamma \vdash e_2 : S_{R_2} \qquad pc \cap R_1 \cap R_2;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash C_1 \\ pc \cap R_1 \cap R_2;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash C_2\end{array}}{pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash \mathbf{if}(e_1 = e_2)\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2} \text{(if\_T)}$$

$$\frac{\begin{array}{c}\Gamma_p;\Gamma_k \vdash RS_1 \qquad \mathbf{pub}(p) \in RS_1 \\ p \in \Gamma_p \qquad \Gamma_p;\Gamma_k;\Gamma \vdash e : Enc\{S\}_{R_2} \qquad RS_1 \subseteq (R_2 \cap pc) \\ pc \cap R_2;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \cup \{x : S_{RS_1}\} \vdash C_1 \\ pc \cap R_2;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash C_2\end{array}}{\begin{array}{c}pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash \\ \mathbf{decrypt}_p\ e\ \mathbf{as}\ x{:}S_{RS_1}\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\end{array}} \text{(dec\_T)}$$

$$\frac{pc = \bot \qquad pc;\Gamma_p;\Gamma_k;\Gamma_c \cup \{c{:}Chan(S_\bot)_\bot\};\Gamma \vdash C}{pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash \mathbf{connect}\ c{:}Chan(S_\bot)_\bot;\ C} \text{(connect\_1\_T)}$$

$$\frac{\begin{array}{c}\Gamma_p;\Gamma_k \vdash R_1 \qquad \Gamma_p;\Gamma_k \vdash R_2 \\ p \in \Gamma_p \qquad k \in \Gamma_k \qquad \{\mathbf{pub}(p), k\} \subseteq R_1 \subseteq R_2 \subseteq pc \\ R_2;\Gamma_p;\Gamma_k;\Gamma_c \cup \{c : Chan(S_{R_1})_{R_2}\};\Gamma \vdash C\end{array}}{pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash \mathbf{connect}\ c{:}Chan(S_{R_1})_{R_2}\ \mathbf{to}\ k\ \mathbf{as}\ p;\ C} \text{(connect\_2\_T)}$$

$$\frac{\begin{array}{c}\Gamma_p;\Gamma_k;\Gamma \vdash e : S_{R_3} \qquad c : Chan(S_{R_1})_{R_2} \in \Gamma_c \\ pc = R_2 \qquad R_1 \subseteq R_3 \qquad pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash C\end{array}}{pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash \mathbf{output}\ c\langle e\rangle;\ C} \text{(output\_T)}$$

$$\frac{\begin{array}{c}c : Chan(S_{R_1})_{R_2} \in \Gamma_c \\ R_2 = pc \qquad pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \cup \{x : S_{R_1}\} \vdash C\end{array}}{pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash \mathbf{input}\ c(x);\ C} \text{(input\_T)}$$

$$\frac{\begin{array}{c}p_2 \in \Gamma_p \qquad \Gamma_p;\Gamma_k;\Gamma \vdash e{:}PrivKeyEnc_\bot \\ pc;\Gamma_p \cup \{p_1\};\Gamma_k;\Gamma_c;\Gamma \vdash C_1 \\ pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash C_2 \qquad pc = \bot\end{array}}{pc;\Gamma_p;\Gamma_k;\Gamma_c;\Gamma \vdash \mathbf{register}_{p_2}\ e\ \mathbf{as}\ p_1\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2} \text{(register\_T)}$$

Fig. 8. Typing rules for main commands

$$\frac{\Gamma_p;\Gamma_k \vdash RS \qquad \Gamma_p;\Gamma_k;\Gamma \vdash e : S_R \qquad RS \subseteq R}{\Gamma_p;\Gamma_k;\Gamma \vdash \mathbf{enc}_{RS}(e) : Enc\{S\}_\bot} \text{(enc\_T)}$$

$$\frac{p \in \Gamma_p}{\Gamma_p;\Gamma_k;\Gamma \vdash \mathbf{release}(p) : PrivKeyEnc_\bot} \text{(release\_T)}$$

$$\frac{\forall i, 1 \le i \le n, \quad p_i \in \Gamma_p \qquad \forall i, 1 \le i \le m, \quad k_i \in \Gamma_k}{\Gamma_p;\Gamma_k \vdash \{\mathbf{pub}(p_1), \ldots, \mathbf{pub}(p_n), k_1, \ldots, k_m\}} \text{(rights\_T)}$$

Fig. 9. Types rules for non standard expressions and rights

variables (new_S) does not consider them: their confidentiality is entirely provided by the type system.

Example 3: We consider the following piece of code in which two new principals are declared, and both Alice and Bob may know the value of y, but only Alice may know the value of x:

$$\begin{array}{ll}\langle \{\} \blacktriangleright & \mathbf{newPrin}\ Alice_{\{\}}; \\ & \mathbf{newPrin}\ Bob_{\{\}}; \\ & \mathbf{new}\ x{:}Int_{\{\mathbf{pub}(Alice)\}} = 5; \\ & \mathbf{new}\ y{:}Int_{\{\mathbf{pub}(Alice),\mathbf{pub}(Bob)\}} = 7; \\ & x\ :=\ y; \\ & \mathbf{if}(x = 1)\ \mathbf{then}\ y\ :=\ 1; \\ \rangle & \end{array}$$

Here the assignment of y to x should be allowed because x is protected by rights more confidential than y. However, in the last line, the value of y (which Bob can read) leaks some information about the value of x (which Bob should not be able to read). Therefore, this is an unsafe command and it cannot be typed.

Types rules for encryption: The type rule for encrypting values (enc_T) verifies that the encrypted value is less confidential than the set of keys used for encryption, it then types the ciphertext as a public value, i.e., encryption removes the type restrictions on a value while ensuring that the encryption provides at least as much protection. We note that if the encrypting key depends on non-public data, then the program counter would not be public, which would ensured that the ciphertext was not stored in a public variable. Hence the use of restricted keys will not leak information.

The corresponding decryption rule (dec_T) verifies that the principal p used to decrypt the cipher is valid ($p \in \Gamma_p$) and is consistent with the rights of the decrypted value. As the knowledge of which keys has been used to encrypt is protected with the rights R_2 of the cipher, the success of the decryption also depends on R_2. Therefore, the program counter has to be at least as high as R_2 when typing the continuation. Finally, as with an assignment, the rule enforces that the created variable does not have a type that is more confidential than the program counter.

Types rules for public channels: Typing rules for public channels ensures that these are only of type public and, when they are used, the program counter is $\bot$.

Example 4: The following system illustrates the use of public channels and encryption:

$$
\begin{array}{l}
\langle \{Alice \mapsto \mathbf{prin}(k_a^+, k_a^-, \{\}), bobPub \mapsto k_b^+\} \blacktriangleright \\
\quad \mathbf{new}\ x{:}Int_{\{Alice,bobPub\}} = 7; \\
\quad \mathbf{connect}\ c : Chan(Enc\{Int\}_\perp)_\perp; \\
\quad \mathbf{output}\ c\langle \mathbf{enc}_{\{\mathrm{pub}(Alice),bobPub\}}(x)\rangle; \\
\quad \mathbf{input}\ c(e); \\
\quad \mathbf{decrypt}_{Alice}\ e\ \mathbf{as}\ xInc{:}Int_{\{\mathrm{pub}(Alice),bobPub\}}\ \mathbf{then} \\
\quad x\ :=\ xInc; \rangle \\
\mid \ \langle \{Bob \mapsto \mathbf{prin}(k_b^+, k_b^-, \{\}), alicePub \mapsto k_a^+\} \blacktriangleright \\
\quad !\ \mathbf{accept}\ c : Chan(Enc\{Int\}_\perp)_\perp; \\
\quad \mathbf{input}\ c(z); \\
\quad \mathbf{decrypt}_{Bob}\ z\ \mathbf{as}\ w{:}Int_{\{\mathrm{pub}(Bob),alicePub\}}\ \mathbf{then} \\
\quad \mathbf{output}\ c\langle \mathbf{enc}_{\{\mathrm{pub}(Bob),alicePub\}}(w+1)\rangle; \rangle
\end{array}
$$

Alice and Bob start off knowing each other's public keys, and Bob provides a service to add one to any number sent by Alice. The variable x is restricted so that only Alice and Bob can know the value. Encrypting this value removes these type protections, so that it can be sent across the public channel c. On Bob's device decrypting with Bob's private key replaces these type protections.

Types rules for secure channels: For secure channels (client and server side), the rule (connect_2_T) enforces that the principal who is creating the channel and the principal being connected to, both have the right to access the data passed over the channel, hence $\{\mathrm{pub}(p), k\} \subseteq R_1$. In order to ensure that the possible side effects caused by using the channel are not more restrictive than the data passed over the channel we need that $R_1 \subseteq R_2$. The $R_2 \subseteq pc$ condition stops side channel leakage to the device receiving the connection at the time the channel is opened. Finally, the program counter is set to R_2 once connected: this ensures both devices have the same program counter. Without this we would have implicit leakage, for instance, one device with a public program counter could wait for a message from a device with a non public program counter, then outputs something on a insecure channel. As the sending of the message may depend on a value protected by the program counter, this would result in a leakage.

We make the strong assumption that the existence of a connection attempt can only be detected by someone with the correct private key to match the public key used to set up the connection. If we assumed a stronger attacker, who could observe all connection attempts, we will need the condition that $pc = \perp$ at least for the client.

The output rule (output_T) has two main restrictions: one which verifies that the device still has the program counter agreed with the corresponding device, and $R_1 \subseteq R_3$ i.e., the type on the channel is no less restrictive than the type of data sent. This is because when this data is received it will be treated as data of type R_1.

For channel creation the restriction on the channel must be at least as restrictive as the program counter. For input and output we must additionally check that the program counter has not become more restrictive, hence requiring that the channel restriction and the program counter are equal, i.e., testing the value of a high level piece of data and then sending data over a lower level channel is forbidden.

Example 5: The different roles of R_1 and R_2 is illustrated in these two programs.

$$
\begin{array}{l}
\quad \mathbf{new}\ x{:}Int_{\{\mathbf{pub}(Alice),bob\}} = 7; \\
\quad \mathbf{accept}\ c{:}Chan(Int_{\{\mathbf{pub}(Alice),bob\}})_{\{\mathbf{pub}(Alice),bob\}} \\
\qquad \mathbf{from}\ bob\ \mathbf{as}\ Alice; \\
\quad \mathbf{if}(x > 10)\{ \\
\quad\ x\ :=\ x+1; \\
\quad\ \mathbf{output}\ c\langle x\rangle; \} \\
\\
\mathbf{new}\ x{:}Int_{\{\mathbf{pub}(Alice),bob\}} = 7; \\
\mathbf{accept}\ c{:}Chan(Int_{\{\mathbf{pub}(Alice),bob\}})_\perp\ \mathbf{from}\ bob\ \mathbf{as}\ Alice; \\
\quad \{\mathbf{if}(x > 10)\{ \\
\quad\ x\ :=\ x+1;\ \}\} \\
\mid \ \mathbf{output}\ c\langle x\rangle;
\end{array}
$$

Both programs aim at sending x to Bob, which is a secret shared by Alice and Bob. In the first case, the sending of x depends on its value: therefore the communication should can only be on a channel with rights $Chan(Int_{\{\mathbf{pub}(Alice),bob\}})_{\{\mathbf{pub}(Alice),bob\}}$. In the other example, even if the value of x is updated due to a parallel thread that has a non public program counter, the sending of x is unconditional.

Note that the language does not have "{*if condition then* C }; C'' structure as this construct would not be safe: if C waits infinitely for a connection then C'' is not executed. However, a *delay* command could be added to help the second program to output x after x has been updated.

Type rules for release and register: The release command is similar to the encryption command except that the rights with which the principal is encrypted are provided by the principal value. Therefore, there is no static check to perform in (release_T). The registration rule (register_T), for the same reason has less checks than (dec_T). However, it does enforce that $pc = \perp$, without which we could get non public rights; revealing a such a none public right would then be an information leak. Removing this restriction, and allowing non public rights, would be possible in a more complex type system but we decide not to do so to keep the type system more understandable.

IV. Example: A Secure Cloud Server

As an extended example we consider a cloud server that provides a data storage service. The motivation of our work is to make it possible to type an open cloud service, without the need for a global PKI neither the need to verify that its users run typed programs, so ensuring that it provides security guarantees to all of its users. The server process in Figure 10 defines an open service which users can connect to and register to store data. This data can be shared with another principal, hence the server takes a pair of public keys, representing the principals, when registering.

To keep the example simple this server accepts up to 3 accounts and denies further registrations. The data for each accounts is stored in the data variable defined in the $RegisterUsers$ process; the restriction set used to type this variable specifies that only the server and the two clients named at registration can have knowledge of this data. Additionally, the server keeps track of how often each account is used (in

$Server \equiv$
new $usage : Array\{Int\}_\perp = \{0,0,0\}$;
new $blocked : Array\{Int\}_\perp = \{0,0,0\}$;
new $nextID : Int_\perp = 0$;
$RegisterUsers \mid CheckUsage$

$RegisterUsers \equiv$
! **accept** $newUsers : Chan(Array\{PubKey\}_\perp)_\perp$;
input $newUsers(client1Client2)$;
let $client1 = client1Client2[0]$ **in**
let $client2 = client1Client2[1]$ **in**
synchronized {
new $accountID : Int_\perp = nextID$;
$nextID = nextID + 1$; }
if $(accountID \leq 2)$ **then**{
new $data : Int_{\{\mathbf{pub}(Server),client1,client2\}} = 0$;
$ServeClient(client1, client2, client1)$
| $ServeClient(client1, client2, client2)$ }

$CheckUsage \equiv$! **synchronized**{
new $total : Int_\perp = usage[0] + usage[1] + usage[2] + 3$;
{ **if**$(usage[0] > total/2)$ **then**
$blocked[0]$:= 1 **else** $blocked[0]$:= 0;
| **if**$(usage[1] > total/2)$ **then**
$blocked[1]$:= 1 **else** $blocked[1]$:= 0;
| **if**$(usage[2] > total/2)$ **then**
$blocked[2]$:= 1 **else** $blocked[2]$:= 0; }

$ServeClient(c1, c2, c3) \equiv$
! **accept** $upload : Chan(Int_{\{Server,c1,c2\}})_\perp$
from $c3$ **as** $Server$;
if $(blocked[accountID] = 0)$ **then**{
input $upload(z)$;
$usage[accountID] = usage[accountID] + 1$;
$data = z$; }
| ! **accept** $dowload : Chan(Int_{\{Server,c1,c2\}})_\perp$
from $c3$ **as** $Server$;
if $(blocked[accountID] = 0)$ **then**{
$usage[accountID] = usage[accountID] + 1$;
output $download\langle data\rangle$; }

Fig. 10. An example server that monitors the clients usage but not their data

$Sender \equiv$
accept $otherPrin : Chan(PubKey_\perp)_\perp$;
input $otherPrin(mobileKey)$;
let $mobile = mobileKey$ **in**
newPrin $Alice_{\{mobile\}}$;
accept $releasedPrin : Chan(PrivKeyEnc_\perp)_\perp$;
output $releasedPrin\langle\mathbf{release}(Alice)\rangle$;
accept $c : Chan(PubKey_\perp)_\perp$;
input $c(bobKey)$;
output $bobKey\langle otherPrin\rangle$;
let $bob = bobKey$ **in**
output $c\langle\mathbf{pub}(Alice)\rangle$;
connect $newUsers : Chan(Array\{PubKey\}_\perp)_\perp$;
output $newUsers\langle\{\mathbf{pub}(Alice), bobKey\}\rangle$;
$Send(Alice, bob, 42)$

$Send(p, k, v) \equiv$
connect $upload : Chan(Int_{\{srvKey,\mathbf{pub}(p),k\}})_\perp$
to $srvKey$ **as** p;
new $sharedSecret : Int_{\{srvKey,\mathbf{pub}(p),k\}} = v$;
output $upload\langle sharedSecret\rangle$;

$Mobile \equiv$
newPrin $Mobile_{\{\}}$;
connect $keyChan : Chan(PubKey_\perp)_\perp$;
output $keyChan\langle\mathbf{pub}(Mobile)\rangle$;
connect $releaseChan : Chan(PrivKeyEnc_\perp)_\perp$;
input $releaseChan(encaps)$
register$_{Mobile}$ $encaps$ **as** $MyId$ **then**
input $keyChan(bobKey)$;
let $bob = bobKey$ **in** $Send(MyId, bob, 24)$

$Receiver \equiv$
newPrin $Bob_{\{\}}$;
connect $fromBob : Chan(PubKey_\perp)_\perp$;
output $fromBob\langle\mathbf{pub}(Bob)\rangle$;
input $fromBob(aliceKey)$;
let $alice = aliceKey$ **in**
connect $download : Chan(Int_{\{srvKey,alice,\mathbf{pub}(Bob)\}})_\perp$
to $srvKey$ **as** Bob;
input $download(data)$;

Fig. 11. Definitions of Sender, Receiver and Mobile Processes

$$
\begin{array}{lcl}
SD & \equiv & \langle\{srvKey \mapsto k_s^+\} \blacktriangleright Sender\rangle \\
MD & \equiv & \langle\{srvKey \mapsto k_s^+\} \blacktriangleright Mobile\rangle \\
RD & \equiv & \langle\{srvKey \mapsto k_s^+\} \blacktriangleright Receiver\rangle \\
Srv & \equiv & \langle\{Server \mapsto \mathbf{prin}(k_s^+, k_s^-, \{\})\} \blacktriangleright Server\rangle \\
System & \equiv & SD \mid MD \mid RD \mid RD \mid Srv
\end{array}
$$

Fig. 12. The server context with 4 devices: a sender with its mobile device and two concurrent receivers

the $usage$ array) and runs a process to monitor the usage (the $CheckUsage$ process). If any account is found to have made more than 50% of the total number of requests (plus 3), it is temporarily blocked (by setting the corresponding index in the $blocked$ array to 1). The usage data and blocked status are public data. This is an example of an open cloud service which writes to public variables after processing private data. Our type system ensures that there is no leakage between the two.

An example configuration is given in Figure 12, with the definitions of the processes provided in Figure 11: this configuration consists of four devices SD, MD and two identical RD devices. We assume that, in the physical world, SD and MD are the laptop and respectively the mobile of Alice while the two other devices RD are owned by Bob and Charlie. In the system definition, Alice's and Bob's devices start off knowing the servers public key, but the server has no knowledge of Alice's and Bob's principals. The mobile device MD first creates a new principal identity and shares the public key to SD. Note that RD could also send its private key to SD at this point which is not the expected behavior. To avoid honest users to establish unwanted connection, a port number mechanism should be added to the connections rules. Once SD receives the principal's public key from MD, SD creates a new principal identity to use with the cloud service which is known by the mobile's principal identity. This allows SD to release and to send the new principal $Alice$ to

MD which registers it. Therefore both SD and MD can use the service with the same account. Finally Bob's device RD and SD exchange their public keys, and SD sends to MD the public key received from RD then SD registers for a shared account between pub($Alice$) and $bobKey$ on the server. Finally, SD or MD can upload a $sharedSecret$ value to the server. Meanwhile MD is able to recover the last uploaded value (0 if it downloads before an upload occurs).

The security type on the variable $sharedSecret$ means that its value can only have an effect on other variables with the same or a more restrictive type. Importantly, our correctness result limits knowledge of these values to just the Alice, Bob and Server devices, no matter what well-typed code are run in these devices. On the other hand, checking the authenticity of the Bob key (with a mechanism such as PGP, or out-of-band checks) is Alice's responsibility.

These are exactly the guarantees that a user, or a organisation, would want before using the cloud service. While many people trust their cloud services, and organisations enter into complex legal agreements, leaks can still occur due to programming errors. Type checking the code, as we outline in this paper can show that it is safe from such programming errors, and help provide the users with the guarantees they need to use the system.

V. Security analysis

We now prove that the type system preserves confidentiality of data: when a variable is declared with rights R then the only devices that can observe anything when the variable's value changes are the devices that are *allowed* to know one of the keys in R.

The proof uses techniques from the applied pi-calculus, rephrased for our formalism. Our basic result uses a notion of bisimulation formulated for reasoning about information flow in nondeterministic programs [18]. Intuitively, two programs are bisimilar (for "low" observers) if each one can "imitate" the low actions of the other, and at each step the memories are equivalent at low locations. Note that memory can change arbitrarily at each step, reflecting the ability of concurrent attacker threads to modify memory at any time.

The applied pi-calculus extends the well-known pi-calculus, a process calculus where communication channels may be sent between processes, with binding of variables to non-channel values. In our approach, "memory" is this set of bindings of variables to values in the applied pi-calculus. Also, our bisimilarity is a labelled bisimilarity since we consider communications on channels as observable events. Our correctness result shows that a high (insider) attacker cannot leak information to low observers by communication on high channels or by modifying high locations in memory.

We explain our proof over the following five subsections. In the following subsection we annotate devices with an identifier, so that we can keep track of particular devices as a process reduces, and we define when a particular device is entitled to read data protected with a particular set of rights. In Subsection V-B we define our untyped attacker and outline an labelled, open semantics which defines how an "honest" (typed) process can interact with an untyped attacker process. We also prove that this open semantics is correct with respect to the semantics presented above.

To give us the machinery we need to prove our main results, in Subsection V-C we annotate our processes with the rights that apply to all variables. We show that a well annotated, well typed process reduces to a well annotated, well typed process, this results shows that a well typed system does not leak data, but it does not account for untyped attackers. To do this we introduce a labelled bisimulation in Subsection V-D. This bisimulation relation defines the attackers view of a process, and their ability to distinguish systems. Finally, in Subsection V-E, we prove that, for our open semantics, a well annotated, well typed process is bisimular to another process that is the same as the first, except that the value of a high level variable is changed. This means that no information can leak about the value of that variable for any possible attacker behaviour, so proving our main correctness results.

A. From rights to allowed devices

As a preliminary step, we need to formally define which devices are and are not granted permissions by a particular set of rights. To do this we need a way to refer to particular devices while they make reductions, so as a notational convention, we place *identifiers* on devices, that are preserved through reductions. By convention an identifier will be an index on each device, so for example $D_1 \mid D_2 \rightarrow D_1' \mid D_2'$ expresses that D_i and D_i' represent the same physical device in different states.

In Definition 1, Definition 2 and Definition 3 below, we formally define an association between the public keys in a rights set and devices, but first we motivate these definitions with an example:

Example 6: Consider the the system $SD_A \mid SD_B \mid MD_M \mid MD_N \mid RD_X \mid RD_Y \mid Srv_S$ where SD, MD, RD and Srv are defined in Figure 11 and Figure 12 (i.e. there are two clones of each devices of the system from Section IV, except for the server). Consider the variable $data$: $Int_{\{\mathbf{pub}(Server),client1,client2\}}$ of the $RegisterUser$ command on Device S (the server). There are three reasons for a device to be allowed to access shared data, depending on which reduction occurs in the system.

First, the devices that created the keys $client1$ and $client2$ are allowed access to this data. This pair of keys is passed to the server at the start of its loop. Depending on which device (A or B) made the connection to the server channel $newUsers$, $client1$ allows either Device A or Device B (as $client1$). Assume that it is Device A. Similarly $client2$ represents Device X or Y depending on which device connected to channel c during the $Sender$ command of Device A. Assume it is Device X.

Next, since the public key $client1$ has been created by the command $\mathbf{newPrin}\ Alice_{\{mobile\}}$ in Device A, the device which corresponds to the public key $mobile$ is also allowed to access $data$. We assume that it is Device M.

Thirdly, the public key $\mathbf{pub}(Server)$ has not been generated by any device. However it was in the initial memory of Device S, therefore this device is also allowed to access the shared data by the right granted by this key.

Our security property grants that no other device than Device S, A, X and M can get information about the value of $data$. On the other hand, if an untyped attacker provides its own key to Device A, through channel c, then no security guarantee can be provided about $data$. Indeed, such a case means that the rights explicitly allows the attacker's device to access the data as any other regular device.

Before we formalise what are the allowed devices, we make reasonable assumption about the initial process. For instance, when the process starts, we assume that devices have not already established any channel between them. We also consider that they have an empty memory except for some public keys and principals (and we do not allow duplicate principals). Finally, we consider that all devices are well-typed except one (Device 0) which is the untyped attacker.

We first define a well-formed and well-typed condition on processes:

Definition 1: A *valid* initial process $P = \nu\bar{c}.\ \langle M_0 \blacktriangleright C_0\rangle_0 \mid \langle M_1 \blacktriangleright C_1\rangle_1 \mid \ldots \mid \langle M_n \blacktriangleright C_n\rangle_n$ is a process where:

1) There is no active channel already established between the processes: $\bar{c} = \{\}$.
2) The bound values in memory are either principals or public keys: For all $0 \leq i \leq n$, $x \mapsto w \in M_i$ implies there exists k^+, $w = \text{prin}(k^+, k^-, \{\})$ or $w = k^+$.
3) Each principal exists only on one device: For all $0 \leq i, j \leq n$, $i \neq j$, $\text{prin}(k^+, k^-, \{\}) \in M_i$ implies $\text{prin}(k^+, k^-, \{\}) \notin M_j$.
4) The memory of every device is well-typed with contexts corresponding to its memory and $pc = \bot$: for all $1 \leq i \leq n$, w.l.o.g. assume that $M_i = \{p_1 \mapsto \text{prin}({k_1}^+, {k_1}^-, \{\}), \ldots, p_m \mapsto \text{prin}({k_m}^+, {k_m}^-, \{\}), pk_1 \mapsto {k'_1}^+, \ldots pk_p \mapsto {k'_p}^+\}$, we have $\bot; \{p_1, \ldots, p_n\}; \{pk_1, \ldots, pk_p\}; \{\}; \{\} \vdash C$ for some command C.

Before defining the set of allowed devices, we define an auxiliary function that maintains which devices are allowed to access shared data, and which public keys need to be associated to devices. This auxiliary metafunction maps backward from a set of rights LR that confers access, to all possible devices that may have provided the keys that gave them those rights. This is the set of allowed devices Entitled$(LR)_T$ where T is a process trace, defined below. Any device that is not in this set is not allowed by LR; we will consider such devices as attacker devices in our threat model.

Definition 2: Given a reduction $P \to P'$ where devices identifiers are in $\{0, \ldots, n\}$, given a subset of identifiers $I \subseteq \{1, \ldots, n\}$ and a set of public keys LR, we define the backward function $\mathcal{B}_{P \to P'}(I, LR)$ in the following way.

- If $P \to P'$ is the reduction (newPrin_S) on a device D_i, $i \neq 0$ that creates a new principal $\text{prin}(k^+, k^-, LR')$ and that $k^+ \in LR$ then

$$\mathcal{B}_{P \to P'}(I, LR) = \left(I \cup \{i\}, LR' \cup LR \setminus \{k^+\}\right).$$

- Otherwise $\mathcal{B}_{P \to P'}(I, LR) = (I, LR)$.

Definition 3: Let $P_0 = \langle M_0 \blacktriangleright C_0\rangle_0 \mid \langle M_1 \blacktriangleright C_1\rangle_1 \mid \ldots \mid \langle M_n \blacktriangleright C_n\rangle_n$ a valid initial process. Let a sequence of reductions $T = P_0 \to P_1 \ldots \to P_n$ and let LR a set of public keys, we consider

$$(I_0, LR_0) = \mathcal{B}_{P_0 \to P_1} \circ \ldots \circ \mathcal{B}_{P_{n-1} \to P_n}(\emptyset, LR).$$

Let $I' = \{i \mid \exists k^+ \in LR_0, i \in \{1, \ldots, n\}, \text{prin}(k^+, k^-, \{\}) \in M_i\}$. We define the set of allowed devices identifiers Entitled$(LR)_T$ as Entitled$(LR)_T = I_0 \cup I'$.

Consider the set $\{k^+ \mid k^+ \in LR_0 \wedge \nexists i \in I', \text{prin}(k^+, k^-, \{\}) \in M_i\}$. We say that Entitled$(LR)_T$ is *safe* if this set is empty.

In other words, Entitled$(LR)_T$ is safe when all keys involved by LR have been either created by devices of Entitled$(LR)_T$ or owned by them at the beginning. This implies, since valid initial processes don't have duplicated keys, that the untyped attacker whose index cannot be in Entitled$(LR)_T$ have not generated any of these keys.

B. Definition of the attacker and of the open process semantics

An attacker is a device $A = \langle M \blacktriangleright C\rangle$ where M is a standard memory and C is a command which is not typed and which contains additional expressions to do realistic operations that an attacker can perform like extracting k^-, k^+ and LR from $\text{prin}(k^+, k^-, LR)$, decrypting a ciphertext with only a secret key, or releasing a principal with arbitrary rights ($\mathbf{enc}_{LR,n}(\text{prin}(k^+, k^-, LR'))$ with $LR \neq LR'$).However, the attacker is not able to create principals with a public key that does not correspond to the private key because we assume that the validity of any pairs is checked when received by an honest device. We denote such an extended expression using E.

To reason about any attacker, we introduce open processes in a similar way as in the applied pi-calculus [1]. An open process has the syntax $\mathcal{K} \models \nu\bar{c}.\ D_1 \mid \ldots \mid D_n$ where $D_1, \ldots, D_n$ are well-typed devices (the indexes $1, \ldots n$ are the tags of the devices: we do not change them through reductions), where $\bar{c}$ are the channels which have been established between devices $D_1, \ldots, D_n$ (not with the attacker) and where $\mathcal{K}$ is a memory representing the values that the attacker already received. We refer to $D_1, \ldots, D_n$ as the *honest devices*. We also refer to $\mathcal{K}$ as the *attacker knowledge*. This plays the same role as frames in the applied pi-calculus, and also plays a similar role as computer memory in the bisimulation that we use for reasoning about noninterference for nondeterministic programs. We denote $\mathcal{K}(E)$ the evaluation of E with the memory $\mathcal{K}$ (to be defined the variables of E should exists in $\mathcal{K}$).

Our type system ensures that an attacker is never able to learn any of the secret keys belonging to "honest" devices, as represented by the notion of reference rights defined below. The following predicate overapproximates what an attacker can learn about a value, based on the "knowledge" represented by its memory $\mathcal{K}$ and on the assumption that it knows all keys which are not in LR (which aims at being the reference rights).

Definition 4: Given a set of keys LR and a value v, we define the predicate $\mathcal{K} \vdash_{LR} v$ as there exists an extended expression E such that $\mathcal{K}(E) = v$, where this extended expression contains all standard functions and attacker functions, as well as an oracle function which provides the secret key of any public key which is not in LR.

Open processes have two forms of reductions: *internal reductions*, which are the same as the reductions for closed processes, and *labeled reductions* which are reductions involving the attacker. Labels on these latter reductions represent the knowledge that an attacker can use to distinguish between two traces, effectively the "low" information that is leaked from the system.

There are two forms of labelled reductions, both of which take the form $P \xrightarrow{l} P'$. In the first form, *input reductions* $P \xrightarrow{\text{in}((c)_{\text{att}},E)} P'$, the attacker provides data, and in the second form, *output reductions* $P \xrightarrow{\text{out}((c)_{\text{att}},x)} P'$, the attacker receives data from an honest device. There are also two further forms of input reductions: those for establishing channels and those which send data. The reduction that establishes a secure channel takes the form:

$$\mathcal{K} \models \nu \bar{c}.D \mid \langle M \blacktriangleright \mathbf{accept}\ c\colon Chan(S_{R_1})_{R_2} \mathbf{from}\ k\ \mathbf{as}\ P; C\rangle_i \xrightarrow{\text{in}((c)_{\text{att}},(Chan(S_{R_1})_{R_2},E,E'))} \mathcal{K} \models \nu \bar{c}.D \mid \langle M \blacktriangleright C\{(c)_{\text{att}}/c\}\rangle_i$$

where $(c)_{\text{att}}$ is any attacker channel name, $\mathcal{K}(E)$ should be the private key corresponding to $M(k)$ and $\mathcal{K}(E')$ should be the public key of $M(p)$. The reduction to establish a public channel is similar but simpler. There is no need for checks on E or E'.

Note that, unlike standard connection establishment where a fresh channel name is added to $\nu\bar{c}$, here the name of the established channel is provided by the attacker and is not added in $\bar{c}$, which is out of the scope of the attacker. However the attacker has to provide channel names in a separate subdomain which prevents it from using an existing honest channel name. The names which are the attacker's channel names are written $(c)_{\text{att}}$. To summarize, a channel name of form $(c)_{\text{att}}$ represents a channel which is established between a device and the attacker, a channel $c \in \bar{c}$ is a channel between two devices (not accessible from the attacker) and a channel name $c \notin \bar{c}$ and not in the attacker's channel domain is just a program variable representing a future channel. Finally, we consider an implicit injection $c \mapsto (c)_{\text{att}}$ from $\bar{c}$ to attacker's channels.

For input reductions that sends data on an established attacker channel, E is an expression of the extended syntax admitting lower level operations that are available to attackers but not to honest devices, as explained above. There is just one rule for output reduction, saying that an attacker can learn from a value output on an attacker channel:

$$\frac{\text{fresh}(x) \qquad M_1(e) = v}{\mathcal{K} \models \nu \bar{c}.D \mid \langle M_1 \blacktriangleright C' \mid \mathbf{output}\ (c)_{\text{att}}\langle e\rangle; C\rangle_i \xrightarrow{\text{out}((c)_{\text{att}},x)} \mathcal{K} \cup (x \mapsto v) \models \nu \bar{c}.D \mid \langle M_1 \blacktriangleright C' \mid C\rangle_i}$$

Example 7: We consider the system consisting of $MD \mid SD$ from Figure 12 running in parallel with an untyped attacker A. The corresponding open process is initially $\{\} \models SD_A \mid MD_M$. Assume that Device M (MD_M) reduces with the attacker instead of SD_A, we have:

$$\begin{array}{ll} \rightarrow & \{\} \models SD_A \mid MD'_M \\ \xrightarrow{\text{in}((c)_{\text{att}},ct)} & \{\} \models SD_A \mid MD''_M \\ \xrightarrow{\text{out}((c)_{\text{att}},x)} & (x \mapsto k^+) \models SD_A \mid MD'''_M \end{array}$$

where $ct = Chan(PubKey_{\perp})_{\perp}$ and Device M has memory $Mobile' \mapsto \text{prin}(k^+, k^-, \{\})$ (a renaming of the local variable $Mobile$ on the device), after the first internal reduction where the principal is created.

After the first reduction where MD_M creates its principal, the attacker establish the connection with Device M: as M is expecting a connection of type ct, the attacker have to provide ct and one of its channel name $(c)_{\text{att}}$. Next, Device M outputs the value of pub$(Mobile)$ on $(c)_{\text{att}}$ which is then stored on the attacker's memory.

The following defines a subprocess of the "honest" devices of a system P, where some (other) devices of that system may be attacker devices. This subprocess of honest devices will be those defined by Entitled$(LR)_T$. In other words, all devices which are not allowed by some key in LR are assumed to be controlled by the attacker.

Definition 5: Given a process $P = \nu\bar{c}.\ D_1 \mid \ldots \mid D_n$ and $\{\mathcal{I}_1, \ldots, \mathcal{I}_m\} \subseteq \{1, \ldots, n\}$, let $\bar{c}'$ the names of channels between devices of $\{\mathcal{I}_1, \ldots, \mathcal{I}_m\}$ we define the subprocess of devices $P\restriction_{\{\mathcal{I}_1,\ldots,\mathcal{I}_m\}} = \nu\bar{c}'.D'_{\mathcal{I}_1} \mid \ldots \mid D'_{\mathcal{I}_m}$ where D'_i is D_i where each channel name $c \in \bar{c} \setminus \bar{c}'$ have been replaced by an attacker-channel name $(c)_{\text{att}}$.

In the following, we will denote by $P \xRightarrow{l_1,\ldots,l_n} P'$ a sequence of reductions $P \rightarrow {}^*P_1 \xrightarrow{l_1} P'_1 \rightarrow {}^*P_2 \ldots P_n \xrightarrow{l_n} P'_n \rightarrow {}^*P'$. We also denote by $P \xrightarrow{l^?} P'$ a reduction which is either $P \rightarrow P'$ or $P \xrightarrow{l'} P'$ for some label l' and by $P \xrightarrow{l^?}{}^? P'$ either $P \xrightarrow{l^?} P'$ or $P = P'$.

The following proposition states that if there is an execution of a system that includes communication with attacker devices, where all communication is local in this closed system, then we can consider subsystem of this, omitting the attacker devices, where communications with attacker devices are modelled by labelled reductions of the form described above. So the attacker devices become part of the context that the devices in the subsystem interact with through a labelled transition system. Such a subsystem may also exclude some of the "honest" devices in the original system, and in that case we treat those excluded devices as attacker devices in the context (since labels on the reductions only model communication with attackers).

Proposition 1: Let $P = \nu\bar{c}.A \mid D_1 \mid \ldots \mid D_n$ where all D_i are well-typed and A is an untyped device. If $P \rightarrow {}^*\nu\bar{c}'.P'$, then for all subset S of $\{1, \ldots, n\}$, there exist $l_1, \ldots, l_m$ and $\mathcal{K}'$ (the attacker knowledge at the end of the execution) such that

$$\mathcal{K} \models P\restriction_S \xRightarrow{l_1,\ldots,l_m} \mathcal{K}' \models P'\restriction_S$$

where $\mathcal{K}$ is a memory which is the union of the memories of A and all D_i with $i \notin S$ (variable names are renamed whenever there is a name conflict).

For instance, the system $SD_A \mid MS_M \mid SD_0$, where SD_0 is part of the attacker, can perform three internal reductions between MS_M and SD_0 where MS_M sends its public key to SD_0. In Example 7, we provided the three reductions of the system $(\{\} \models SD_A \mid MS_M \mid SD_0)\restriction_{\{A,M\}}$.

C. Extended syntax with extra annotations

In this section, we add extra annotations to processes to perform a specific analysis about a given right $LR = \{pk_1, \ldots, pk_n\}$. Since the keys in LR do not necessary exist in the initial process, we first annotate open processes with a *reference right* $\mathtt{R}_r$ with the intention that this right will eventually grow to the right LR as keys are generated and added to this right during execution. Given a reference right $\mathtt{R}_r$, we define any rights whose all keys are in $\mathtt{R}_r$ to be *high* (by opposition to *low*). The set $\mathtt{R}_r$ starts with keys that exists in the initial process.

An *annotated process* has the syntax

$$\mathcal{K}; \mathtt{R}_r \models P$$

for attacker knowledge $\mathcal{K}$, reference right $\mathtt{R}_r$ and process P. The exact form of the annotations is explained below. The reference right $\mathtt{R}_r$ only changes during a reduction of a command $\mathbf{newPrin}\ p_{RS}$. In this case, there is a choice of whether or not to include the generated public key in $\mathtt{R}_r$.

Definition 6: A sequence of reductions is called *standard* if each time a (newPrin_S) reduction adds a key to $\mathtt{R}_r$:

$$\mathcal{K}; \mathtt{R}_r \models \nu \bar{c}.\langle M \blacktriangleright \mathbf{newPrin}\ p_{RS}; C\rangle \mid P$$

$$\rightarrow \mathcal{K}; \mathtt{R}_r \cup \{k^+\} \models \nu \bar{c}.\langle M \cup p \mapsto \mathrm{prin}(k^+, k^-, LR) \blacktriangleright C\rangle \mid P$$

we have $LR \subseteq \mathtt{R}_r$.

The next proposition ensures the existence of a standard annotation such that the rights we want to consider at the end are high and that the attacker does know any key in the set $\mathtt{R}_r$ of the initial process. We will state in Proposition 3 that this implies that the attacker never knows the keys in $\mathtt{R}_r$ during the whole reduction.

Proposition 2: Let P be a valid initial process such that $P \rightarrow {}^*P'$ and let LR a set of keys defined in P'. If $D_H = \mathrm{Entitled}(LR)_{P \rightarrow^* P'}$ is safe then there exists a standard annotation for the reduction provided by Proposition 1: $\mathcal{K}_0; \mathtt{R}_{r0} \models P \restriction_{D_H} \xrightarrow{l_1,\ldots,l_n} \mathcal{K}; \mathtt{R}_r \models P' \restriction_{D_H}$ where $\mathtt{R}_{r0}$ and $\mathtt{R}_r$ are such that $\forall k^+ \in \mathtt{R}_{r0},\ \mathcal{K}_0 \nvdash_{\mathtt{R}_{r0}} k^-$ and $LR \subseteq \mathtt{R}_r$.

When a variable or a channel is created with some rights, the reduction rules of the operational semantics remove all information about those rights. Therefore we add annotations to devices to remember if the defined right was high or low, according to $\mathtt{R}_r$. Recall that a right is high if it contains a key in the reference right $\mathtt{R}_r$. We use ℓ as a metavariable for an annotation $\mathcal{H}$ or $\mathcal{L}$.

1) A memory location $x \mapsto v \in M$ which is created by a command $\mathbf{new}\ x : S_R = e; C$, where R evaluates in M to a high right according to $\mathtt{R}_r$, is annotated as a high location: $x^{\mathcal{H}} \mapsto v$. Otherwise x is annotated as a low location: $x^{\mathcal{L}} \mapsto v$. By convention $\mathcal{K}$ contains only low locations.
2) A new channel name c, which is created by a command that establishes a channel $c : Chan(S_{R_1})_{R_2}$, is annotated as $c^{\ell_1}_{\ell_2}$ where ℓ_1 resp. ℓ_2 is $\mathcal{H}$ if the evaluation of R_1 (resp. R_2) is $high$, and is annotated as $\mathcal{L}$ otherwise.
3) A value which is the result of an expression (besides an encryption expression) where one variable refers to a high location is annotated as a high value $(v)^{\mathcal{H}}$. Otherwise, it is annotated as a low value $(v)^{\mathcal{L}}$. When a value $(v)^{\ell}$ is encrypted, it becomes $(\mathbf{enc}_{RS,n}((v)^{\ell}))^{\mathcal{L}}$ i.e., the initial tag is associated to the subterm.
4) A device $\langle M \blacktriangleright C_1 \mid \ldots \mid C_n\rangle$ is annotated as $\langle M \blacktriangleright (C_1)_{(\ell_1)} \mid \ldots \mid (C_n)_{(\ell_n)}\rangle_i$. There is one tag $\ell_i \in \{\mathcal{L}, \mathcal{H}\}$ for each sub-command C_i which can be reduced. The annotation of each thread $(C_i)_{(\ell_i)}$ records whether the existence of this thread was due to a high value or a high right. When $\ell_i = \mathcal{H}$, we say that the thread is *high*, otherwise the thread is *low*. For instance, the annotated device $\langle a^{\mathcal{H}} \mapsto (1)^{\mathcal{H}} \blacktriangleright (\mathbf{if}\ (a = 0)\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2)_{(\mathcal{L})}\rangle_1$ reduces to $\langle a^{\mathcal{H}} \mapsto 1 \blacktriangleright (C_2)_{(\mathcal{H})}\rangle_1$ because a is a high location. The other case where a thread can be set to high is in the establishment of a secure channel that is annotated with $c^{\mathcal{H}}_{(\mathcal{H})}$.

These annotations allow us to define technical invariants that are preserved during reduction. In the following technical definition, we formalize the idea that there exist a typing judgment for devices (Case (1) below) which is consistent with the annotations:

1) Variables (Case (2)) and channels (Case (3)) should have a type which corresponds to their annotation
2) The type system tracks a notion of security level for the control flow, and this level pc must be consistent with the thread annotation (Case (4)).
3) In addition, we express that the devices are not in a corrupted configuration, in the sense that secure channels are not being used to communicate with the attacker (Case (5)), nor are they used in a low thread (Case (6)).
4) Finally, ciphers for values (Case (7)) and principals (Case (8)) should not have high contents protected by low keys.

Definition 7: A tuple consisting of a reference right $\mathtt{R}_r$, a memory M and a thread $(C)_{(\ell)}$ is *well-annotated* written "$\mathtt{R}_r \Vdash M \rhd (C)_{(\ell)}$: *well-annotated*" if there exists pc, Γ_p, Γ_k, Γ_c and Γ such that

1) $pc; \Gamma_p; \Gamma_k; \Gamma_c; \Gamma \vdash C$
2) For all locations x in M, we have
   - either $x^{\mathcal{L}} \mapsto (v)^{\mathcal{L}}$ in M for some value v and $x \in \Gamma_p \cup \Gamma_k$
   - or $\Gamma_p; \Gamma_k; \Gamma \vdash x : S_R$ and
     - if $\mathtt{R}_r \supseteq M(R)$, $x^{\mathcal{H}} \mapsto (v)^{\mathcal{H}}$ in M,
     - if $\mathtt{R}_r \not\supseteq M(R)$, $x^{\mathcal{L}} \mapsto (v)^{\mathcal{L}}$ in M for some value v.
3) For all channels c in the thread such that $\Gamma_c \vdash c : Chan(S_{R_1})_{R_2}$, the annotation of c is $c^{\ell_2}_{(\ell_1)}$ where $\ell_2 = \mathcal{H}$ iff. $\mathtt{R}_r \supseteq R_2$, $\ell_1 = \mathcal{H}$ iff. $\mathtt{R}_r \supseteq M(R_1)$.
4) $\mathtt{R}_r \supseteq pc$ if and only if $\ell = \mathcal{H}$.
5) For all $(c)_{\text{att}}$, we have $\Gamma_c \vdash (c)_{\text{att}} : Chan(S_{R_1})_{R_2}$ with $\mathtt{R}_r \not\supseteq M(R_2)$ and $\mathtt{R}_r \not\supseteq M(R_1)$.
6) if ℓ is $\mathcal{L}$, then there is no $c^{\mathcal{H}}_{(\mathcal{H})}$ in C.

7) For all values v stored in memory, if a sub-term of v matches $\mathbf{enc}_{LR,n}((t)^{\mathcal{H}})$ then $\mathtt{R}_r \supseteq LR$.
8) For all values v in memory, if a sub-term of v matches $\mathbf{enc}_{LR',n}(\mathbf{prin}(k, LR,))$ then $LR = LR'$ or $\mathtt{R}_r \supseteq LR$.

When, for a device $D = \langle M \blacktriangleright (C_1)_{(\ell_1)} \mid \ldots \mid (C_n)_{(\ell_n)} \rangle_i$ and a set $\mathtt{R}_r$, we have $\mathtt{R}_r \Vdash M \rhd (C_i)_{(\ell_i)}$: well-annotated for $i \in \{1, \ldots, n\}$, then we use the notation $\mathtt{R}_r \Vdash D$: well-annotated.

Finally, we get the following subject-reduction result:

Proposition 3: Let $\mathcal{K}; \mathtt{R}_r \models \nu\bar{c}.D_1 \mid \ldots \mid D_n$ an open process such that for all $k^+ \in \mathtt{R}_r$ we have $\mathcal{K} \nvdash_{\mathtt{R}_r} k^-$ and for all $1 \leq i \leq n$ we have $\mathtt{R}_r \Vdash D_i$: well-annotated. If $\mathcal{K}; \mathtt{R}_r \models \nu\bar{c}.D_1 \mid \ldots \mid D_n \xrightarrow{l?} \mathcal{K}'; \mathtt{R}_r{}' \models \nu\bar{c}'.D'_1 \mid \ldots \mid D'_n$ then for all $1 \leq i \leq n$ we have $\mathtt{R}_r \Vdash D'_i$: well-annotated. Moreover if the reduction is standard, we have $\mathcal{K}' \nvdash_{\mathtt{R}_r{}'} k^-$ for all $k^+ \in \mathtt{R}_r{}'$.

Finally, valid initial processes which only require each device to be well-typed are well-annotated.

Proposition 4: Given a valid initial process $P = \Big(\mathcal{K} \models \langle M_1 \blacktriangleright C_1 \rangle_1 | \ldots | \langle M_n \blacktriangleright C_n \rangle_n \Big)$, We have $\mathtt{R}_r \Vdash M_i \rhd (C_i)_{(\mathcal{L})}$: well-annotated for $1 \leq i \leq n$ for any $\mathtt{R}_r$.

D. Labelled bisimilarity

The invariants expressed above are about a single process. To ensure that the attacker cannot track implicit flows, we need to compare the execution of two processes in parallel. In this section, we define a relation between processes which implies an adapted version of the bisimilarity property of the applied pi-calculus [1].

The two processes that we compare are the actual process and another one where the value of one of the high variables of the memory of some device has been changed to another one. So first, we define what is a process where a variable is modified arbitrarily.

Definition 8: Given a device $D = \langle M \cup \{x \mapsto v'\} \blacktriangleright C \rangle$, and a value v, we define $D_{x=v}$ to be the device that updates the variable x to be v by assignment, $\langle M \cup \{x \mapsto v'\} \blacktriangleright C \mid x := v; \rangle$. Extending this from devices to processes, given a process P which contains D with index i, we define $P_{i:x=v}$ to be the same process except that D has been replaced by $D_{x=v}$.

In contrast to systems where the attacker can only observe public values in memory after reduction, here we model that the attacker can observe communications on channels that have been established with other devices. In the following examples, we stress how an attacker can distinguish between two processes even without knowing actual confidential values in the memory of those devices.

Example 8: We consider the device $D(X, Y)$, where the description is parametrized by two meta-variables: X is a value stored in memory and Y is a value that is encrypted and sent on an attacker channel. The device has a memory $M(X) = \{x \mapsto X, k \mapsto {k_0}^+\}$, and the full description of the device is

$$\langle M(X) \blacktriangleright \mathbf{if}\ x = 0\ \mathbf{then\ output}\ (c)_{\text{att}} \langle \mathbf{enc}_k(Y) \rangle \rangle_A.$$

The process $D(0, Y)$ can be distinguished from the process $D(1, Y)$. In the first case, we have:

$$\{\} \models D(0, Y) \xrightarrow{\text{out}((c)_{\text{att}}, m)}$$
$$\{m \mapsto \mathbf{enc}_k(Y)\} \models \langle M(X) \blacktriangleright\ \mathbf{skip}\ \rangle_A$$

On the other hand, there is no reduction with only the label $\text{out}((c)_{\text{att}}, m)$ and internal reductions starting from $\{\} \models D(1, Y)$. This distinction models the fact that if the attacker receives data on $(c)_{\text{att}}$, it learns that $X = 0$.

The processes $D(0, 2)$ and $D(0, 3)$ can also be distinguished even if both $\{k \mapsto {k_0}^-\} \models D(0, 2)$ and $\{k \mapsto {k_0}^-\} \models D(0, 3)$ can reduce with a label $\text{out}((c)_{\text{att}}, m)$. Indeed, after reduction the attacker's knowledge is $\mathcal{K} = \{k \mapsto {k_0}^-, m \mapsto \mathbf{enc}_{{k_0}^+,n}(Y)\}$ where Y is 2 (resp. 3): By performing the decryption of m with an untyped decryption, the attacker can compare the result to 2, and $\mathcal{K}(\text{decr}_k(m)) = 2$ is only true in the first case.

Finally, the processes $\{\} \models D(0, 2)$ and $\{\} \models D(0, 3)$ are not distinguishable. In both cases:

- there is no test to distinguish between the two attacker's knowledge, and
- the labelled reductions are the same i.e $\text{out}((c)_{\text{att}}, m)$

With these examples in mind, we introduce static equivalence, an adaptation of the one used in the applied pi-calculus, which expresses that it is not possible to test some equality which would work with one memory but not with the other.

Definition 9: We say that two memories M_1 and M_2 are statically equivalent ($\approx_s$) if they have exactly the same variable names and for all extended expressions E where its variables $x_1, \ldots, x_n \in M_1$, we have $M_1(E) = M_2(E)$.

Finally, we define an adaptation of the labelled bisimilarity. This recursive definition generalizes the conclusion of the example: the two memories should be statically equivalent, and a transition in one process can be mimicked in the other process, where the reduced processes should also be bisimilar.

Definition 10: Labeled bisimilarity ($\approx_l$) is the largest symmetric relation $\mathcal{R}$ on open processes such that $(\mathcal{K}^A \models P^A)\ \mathcal{R}\ (\mathcal{K}^B \models P^B)$ implies:

1) $\mathcal{K}^A \approx_s \mathcal{K}^B$;
2) if $(\mathcal{K}^A \models P^A) \rightarrow (\mathcal{K}^{A\prime} \models P^{A\prime})$, then $(\mathcal{K}^B \models P^B) \rightarrow {}^*(\mathcal{K}^{B\prime} \models P^{B\prime})$ and $(\mathcal{K}^{A\prime} \models P^{A\prime})\ \mathcal{R}\ (\mathcal{K}^{B\prime} \models P^{B\prime})$ for some $\mathcal{K}^{B\prime} \models P^{B\prime}$;
3) if $(\mathcal{K}^A \models P^A) \xrightarrow{l} (\mathcal{K}^{A\prime} \models P^{A\prime})$, then $(\mathcal{K}^B \models (\mathcal{K}^B \models P^B) \stackrel{l}{\Longrightarrow} (\mathcal{K}^{B\prime} \models P^{B\prime})$ and $(\mathcal{K}^{A\prime} \models P^{A\prime})\ \mathcal{R}\ (\mathcal{K}^{B\prime} \models P^{B\prime})$ for some $\mathcal{K}^{B\prime} \models P^{B\prime}$.

The labelled bisimilarity is a strong equivalence property, and its we use it to state the security property that an attacker is unable to distinguish between two processes in the same class [2]. However this definition does not help to actually compute the processes in the same class. Therefore we define a stronger relation that can be defined from our annotated semantics.

First, we define an obfuscation function $\text{obf}(LR, w)$ which takes a set of public key LR and a value or a principal w and

returns an obfuscated value (its syntax is like the syntax of value except that there is an additional option ■ and that a principal value is also an option).

Definition 11: Let LR a set of public key, w a principal p or a value v. We define $\mathrm{obf}(LR, w)$ depending on the structure of w.

$$\mathrm{obf}(LR, \mathbf{enc}_{LR',n}(w')) = \begin{cases} \mathbf{enc}_{LR',n}(\mathrm{obf}(LR, w')) \text{ if } LR \supsetneq LR' \\ \mathbf{enc}_{LR',n}(\blacksquare) \text{ if } LR \supseteq LR' \end{cases}$$

$$\mathrm{obf}(LR, \{v_1, \ldots, v_n\}) = \{\mathrm{obf}(LR, v_1), \ldots, \mathrm{obf}(LR, v_n)\}$$

$$\text{otherwise } \mathrm{obf}(LR, w) = w.$$

where w' is a value or a principal and n a nonce.

The set LR aims at containing a set of keys whose private keys will never be known by the attacker. In our previous example, we have $\mathrm{obf}(k_0{}^+, \mathbf{enc}_{k_0{}^+,n}(Y)) = \mathbf{enc}_{k_0{}^+,n}(\blacksquare)$: if $k_0{}^-$ is never known by the attacker, the attacker will never be able to know anything about Y. Note that the random seed n used for the cipher is not hidden. Indeed, the attacker is able to distinguish between a cipher that is sent twice and two values which are encrypted then sent: in the first case the two message are strictly identical.

We now define an equivalence on memories: given a reference right $\mathtt{R}_r$, a set of safe keys, then we say that two memories are equivalent if they differ only by the high values which aims at never been sent to the attacker and by the obfuscated terms of the low values.

Definition 12 (equivalent memories): Equivalent memory is a relation on memories such that $M_1 \mathcal{R}_{\mathtt{R}_r} M_2$ if for each low location $x^{\mathcal{L}} \mapsto v^A$ in M_1 (resp. M_2), there exists $x^{\mathcal{L}} \mapsto v^B$ in M_2 (resp. M_1) such that $\mathrm{obf}(\mathtt{R}_r, v^A) = \mathrm{obf}(\mathtt{R}_r, v^B)$ and each location $p \mapsto \mathrm{pv}$ in M_1 (resp. M_2) exists in M_2 (resp. M_1).

Next, we define an equivalence relation between two well-annotated processes: two processes are equivalent if they have the same commands up to some additional high threads and have equivalent memories:

Definition 13: Two annotated processes P^A and P^B are *annotated-equivalent* ($P^A \mathcal{R} P^B$) when there exists an alpha-renaming (capture-avoiding renaming of bound variables) of P^B such that

$$P^A \text{ is } \mathcal{K}^A; \mathtt{R}_r \models \nu\bar{c}.\langle M_1^A \blacktriangleright C_{(1,1)}^A \mid \ldots \mid C_{(1,m_1)}^A\rangle_1 | \ldots | \langle M_n^A \blacktriangleright C_{(n,1)}^A \mid \ldots \mid C_{(n,m_n)}^A\rangle_n,$$

$$P^B \text{ is } \mathcal{K}^B; \mathtt{R}_r \models \nu\bar{c}'.\langle M_1^B \blacktriangleright C_{(1,1)}^B \mid \ldots \mid C_{(1,m_1)}^B\rangle_1 | \ldots | \langle M_n^B \blacktriangleright C_{(n,1)}^B \mid \ldots \mid C_{(n,m_n)}^B\rangle_n,$$

and we have $\mathcal{K}^A \mathcal{R}_{\mathtt{R}_r} \mathcal{K}^B$, for all $k^+ \in \mathtt{R}_r$, we have $\mathcal{K}^A \not\vdash_{\mathtt{R}_r} k^-$ and furthermore for all (i, j), first $M_i^A \mathcal{R}_{\mathtt{R}_r} M_i^B$, next, the annotation on each thread $C_{(i,j)}^X$ (where X stands for A or B) is either $\mathcal{H}$ or $\mathcal{L}$ such that either:

- $\mathtt{R}_r \Vdash M_i^X \rhd (C_{(i,j)}^X)_{(\mathcal{H})}$: well-annotated
- $\mathtt{R}_r \Vdash M_i^X \rhd (C_{(i,j)}^X)_{(\mathcal{L})}$: well-annotated and $C_{(i,j)}^A = C_{(i,j)}^B$ (the commands are syntactically identical up to renaming of bound variables).

Finally, we prove that this relation actually implies bisimilarity.

Proposition 5: Let P^A be $(\mathcal{K}^A; \mathtt{R}_r \models \nu\bar{c}.Q^A)$ and P^B be $(\mathcal{K}^B; \mathtt{R}_r \models \nu\bar{c}'.Q^B)$, where both are well-annotated processes such that $P^A \mathcal{R} P^B$, Furthermore assume that for all $k^+ \in \mathtt{R}_r$ we have $\mathcal{K}^A \not\vdash_{\mathtt{R}_r} k^-$. Then the processes are annotated equivalent, $P^A \approx_l P^B$ (with the annotations removed).

E. Main theorems

Our first security property grants confidentiality for each created variable in the following way. When a variable x is created with a protection R, this implicitly defines a set of devices which are allowed to access x. If the untyped attacker is not in this set, then any collaboration of the attacker with the denied-access devices can not learn any information about the value stored by the variable: they cannot detect an arbitrary modification of the variable.

Theorem 1: Let $P = A \mid D_1 \mid \ldots \mid D_n$ be a valid initial process. We consider a reduction $P \rightarrow {}^*P'$ with $P' = \nu\bar{c}.A' \mid D_1' \mid \ldots \mid D_n'$ such that for some $1 \leq i \leq n$, D_i' is $\langle M \blacktriangleright \mathbf{new}\ x : S_R = E; C \mid C'\rangle$. Let $LR = M(R)$ be the set of keys corresponding to R, let $D_H = \mathrm{Entitled}(LR)_{P \rightarrow P'}$ and let $\mathcal{K}$ be the unions of memories of the devices of P' whose indexes are not in D_H and of device A'. If D_H is safe (as stated in Definition 3), then $(\mathcal{K} \models P' \restriction_{D_H}) \approx_l (\mathcal{K} \models P'_{i:x=v} \restriction_{D_H})$.

Proof: According to Proposition 1, we have $\mathcal{K}_0 \models P \restriction_{D_H} \xrightarrow{l_1,\ldots,l_n}{}^* \mathcal{K}' \models P' \restriction_{D_H}$ where $\mathcal{K}_0 = \bigcup_{i \notin D_H} M_i$. As P is a valid initial process, we get that $P \restriction_{D_H}$ is also a valid initial process (the verification of all conditions to be a valid initial process is immediate). Since D_H is safe, we consider $\mathtt{R}_{r0}$, $\mathtt{R}_r$ and the standard annotated semantics from Proposition 2:

$$\mathcal{K}_0; \mathtt{R}_{r0} \models P \restriction_{D_H} \xrightarrow{l_1,\ldots,l_n} \mathcal{K}; \mathtt{R}_r \models P' \restriction_{D_H}$$

where $LR \subseteq \mathtt{R}_r$ and

$$\forall k^+ \in \mathtt{R}_{r0},\ \mathcal{K}_0 \not\vdash_{\mathtt{R}_{r0}} k^-. \tag{1}$$

From Proposition 4, the annotation of the initial process is well-annotated. From Proposition 3 on this annotation, we get for all $i \in D_H$: $\mathtt{R}_r \Vdash D_i$: well-annotated and, due to (1), for all $k^+ \in \mathtt{R}_r$ we have $\mathcal{K} \not\vdash_{\mathtt{R}_r} k^-$. Since $LR \subseteq \mathtt{R}_r$, and given the (assign_T) rule, we also have

$$\mathtt{R}_r \Vdash M \cup \{x^{\mathcal{H}} \mapsto v'\} \rhd (x\ :=\ v)_{(\mathcal{H})} : \text{well-annotated.}$$

Therefore, we satisfy all conditions of Definition 13:

$$(\mathcal{K}; \mathtt{R}_r \models P') \mathcal{R} (\mathcal{K}; \mathtt{R}_r \models P'_{i:x=v}).$$

Finally, we conclude with Proposition 5. ■

Example 9: From the example of Section IV, let consider the variable $sharedSecret : Int_{\{serverKey, Alice, Bob\}}$ of the device SD. The Theorem 1 states that only devices from $\mathrm{Entitled}(\{serverKey, Alice, Bob\})_{System \rightarrow System'}$ can distinguish between $sharedSecret = 42$ or $sharedSecret = 43$. Whatever are the reductions, these devices contains for sure Srv since its key is known from the beginning and SD since it creates $Alice$. The only threat is that the channel $otherPrin$ has not been established with MD or that c has not been established with the device RD of Bob. These threats could be

removed by an additional authentication protocol or the use of a private channel (for instance if the connection between SD and MD is a direct wired connection). This means that not knowing which code is run on Srv and RD is not a threat as long as Srv and RD guarantee that their codes are well-typed.

Finally, we state a standard security property in the simpler case where there is no untyped attacker. Given a process P, if, at some point, we change the value that a variable is bound to in memory, where that variable has been typed with right R, then new reductions will not alter values in memory locations that have been typed strictly less confidential than R (i.e., those variables have rights that contain public keys not contained in R).

Theorem 2: Let P be a valid initial process. We consider a reduction $P \to P'$ with $P' = D'_1 | \ldots | D'_n$ such that for some i, D'_i is $\langle M \blacktriangleright \mathbf{new}\ x : S_R = E; C \mid C' \rangle$. Let v be any value of type S. Let $P'' = D''_1 | \ldots | D''_n$ such that $P' \to {}^*P''$ then there exists Q'' such that $P'_{x=v} \to {}^*Q''$ where for all memory location y that have been created in any device with rights R_l with $R \supsetneq R_l$, we have $\mathrm{obf}(R, v_P) = \mathrm{obf}(R, v'_Q)$ where v_P is the value of y in P'' and v_Q the value of y in Q''.

Implementation: We have implemented an interpreter for this language in Ocaml[2]. The program strictly follows the extended semantics, it has commands to add a new device; to do an internal reduction on the selected thread(s) of the selected device(s); to perform an attacker communication, and to type-check each device of the system according to the annotations. To define a device which starts with keys in memory, two additional commands are provided which are allowed only as a preamble: $\mathbf{load\ principal}\ p\ \mathbf{from}\ i$; and $\mathbf{load}\ x\ :\ PubKey\ \mathbf{from}\ i$; where i is a number; an identical i on two devices represents a shared key. This implementation demonstrates that the syntax is well-defined and effective, and allows us to test the invariance of the properties with demonstrative examples. The example of Section IV has been tested using this implementation: we are able to reduce the process such that Bob gets the secret from Alice and we can verify that each step correctly type-checks.

VI. Conclusion

We have presented a security type system that provides location based correctness results as well as a more traditional non-interference result. The key novelty of our system is to allow principal identities to be created and shared across devices in a controlled way, without the need for a global PKI or middleware layer. Hence, our correctness result states that well-typed devices can control which other devices may acquire their data, even in the presents of untyped attackers. We have illustrated our system with an example of an open cloud server that accepts new users. This server does perform some monitoring of its users but our type system proves that it does not monitor the content of their data. We argued that our framework is particularly appropriate to cloud systems where organizations will want guarantees about where their data will be stored as well as the secrecy of their data.

Acknowledgement: We would like to thank Alley Stoughton for her help with this work; her insightful comments and useful advice greatly improved this paper.

[2]https://github.com/gazeaui/secure-type-for-cloud

References

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proc. 28th ACM Symposium on Principles of Programming Languages (POPL'01)*, London, UK, 2001. ACM.

[2] Myrto Arapinis, Jia Liu, Eike Ritter, and Mark Ryan. *Stateful Applied Pi Calculus*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[3] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. Cryptographically-masked flows. *Theor. Comput. Sci.*, 402(2-3), 2008.

[4] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2), 2011.

[5] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proc. of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*. IEEE Computer Society Press, 2001.

[6] Tom Chothia and Dominic Duggan. Type-based distributed access control vs. untyped attackers. In *Formal Aspects in Security and Trust, Third International Workshop, FAST*, 2005.

[7] Tom Chothia, Dominic Duggan, and Jan Vitek. Type-based distributed access control. In *16th IEEE Computer Security Foundations Workshop (CSFW-16)*, 2003.

[8] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7), July 1977.

[9] Cedric Fournet, Andy Gordon, and Sergio Maffeis. A type discipline for authorization in distributed systems. Technical Report MSR-TR-2007-47, Microsoft Research, April 2007.

[10] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, 2008.

[11] Ivan Gazeau, Tom Chothia, and Dominic Duggan. Types for location and data security in cloud environments. Technical Report CS-2017-1, Stevens Institute of Technology, Hoboken, NJ, June 2017. Available at https://github.com/gazeaui/secure-type-for-cloud.

[12] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. AURA: a programming language for authorization and audit. In *13th ACM SIGPLAN International Conference on Functional Programming, ICFP*, 2008.

[13] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[14] Jed Liu and Andrew C. Myers. Defining and enforcing referential security. In *Principles of Security and Trust: Third International Conference, POST*, 2014.

[15] Jamie Morgenstern and Daniel R. Licata. Security-typed programming within dependently typed programming. In *15th ACM SIGPLAN International Conference on Functional Programming*, ICFP, 2010.

[16] Stefan Muller and Stephen Chong. Towards a practical secure concurrent language. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications*, 2012.

[17] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4), October 2000.

[18] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Computer Security Foundations Workshop (CSFW'00)*, 2000.

[19] Jeffrey A. Vaughan, Limin Jia, Karl Mazurak, and Steve Zdancewic. Evidence-based audit. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF-08,*, 2008.

[20] Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *IEEE Symposium on Security and Privacy*, 2007.

[21] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08. USENIX Association, 2008.

[22] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference (extended abstract). In *IFIP TC1 WG1.7 Workshop on Formal Aspects in Security and Trust (FAST)*, 2005.